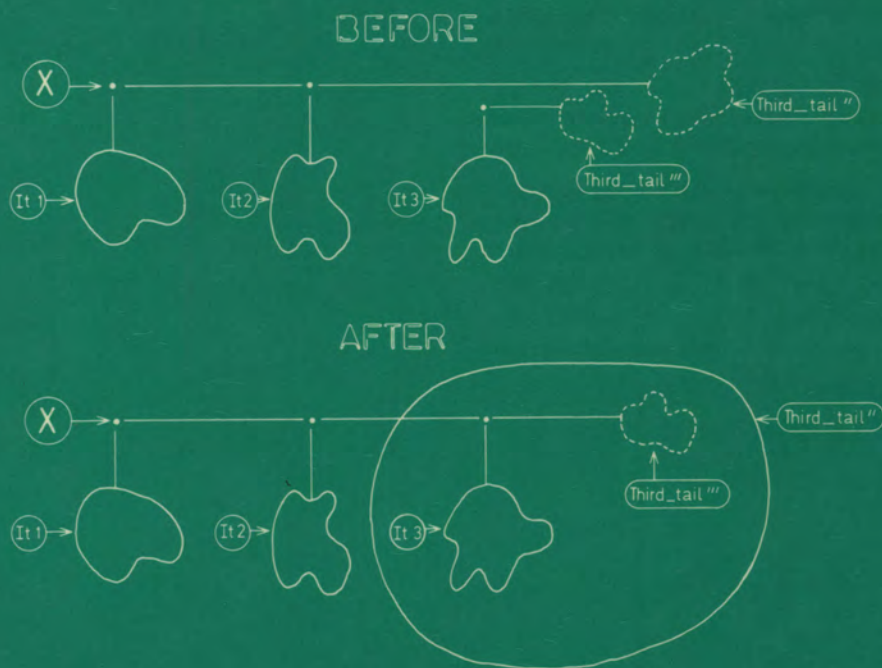


# PROLOG FOR PROGRAMMERS

Feliks Kluźniak  
Stanisław Szpakowicz

With a contribution by  
Janusz S. Bień



---

## **PROLOG FOR PROGRAMMERS**

---



*This is volume 24 in A.P.I.C. Studies in Data Processing  
General Editors: Fraser Duncan and M. J. R. Shave  
A complete list of titles in this series appears at the end of this volume*

---

# PROLOG FOR PROGRAMMERS

---

**Feliks Kluźniak**

**Stanisław Szpakowicz**

*Institute of Informatics  
Warsaw University  
Warsaw, Poland*

*With a contribution by Janusz S. Bień*

1985



**ACADEMIC PRESS**

(Harcourt Brace Jovanovich, Publishers)

London Orlando San Diego New York Austin  
Toronto Boston Sydney Tokyo



COPYRIGHT © 1985, BY ACADEMIC PRESS INC. (LONDON) LTD.  
ALL RIGHTS RESERVED.

NO PART OF THIS PUBLICATION MAY BE REPRODUCED OR  
TRANSMITTED IN ANY FORM OR BY ANY MEANS, ELECTRONIC  
OR MECHANICAL, INCLUDING PHOTOCOPY, RECORDING, OR  
ANY INFORMATION STORAGE AND RETRIEVAL SYSTEM, WITHOUT  
PERMISSION IN WRITING FROM THE PUBLISHER.

REPRINTED WITH CORRECTIONS 1987

ACADEMIC PRESS INC. (LONDON) LTD.  
24-28 Oval Road  
LONDON NW1 7DX

*United States Edition published by*  
ACADEMIC PRESS, INC.  
Orlando, Florida 32887

#### British Library Cataloguing in Publication Data

Kluźniak, Feliks

Prolog for programmers.

1. Programming languages (Electronic computers)
2. Electronic digital computers--Programming

I. Title II. Szpakowicz, Stanisław  
000.64'24 QA76.7

#### Library of Congress Cataloguing in Publication Data

Kluźniak, Feliks.

Prolog for programmers.

Includes index.

1. Prolog (Computer program language) I. Szpakowicz,  
Stanisław. II. Bień, Janusz St. III. Title.

QA76.73.P76K58 1985 001.64'24 84-14520

ISBN 0-12-416520-6 (hardback)

0-12-416521-4 (paperback)

PRINTED IN GREAT BRITAIN  
AT THE ALDEN PRESS, OXFORD



*Agacie Sarze, i jej mamie—F. K.*

*Pakowi, Mikołajowi, Błażejowi i Gumce—Szp.*

*Zosi i Basi—J. S. B.*

---

# CONTENTS

---

<b>PREFACE</b>	xi
<b>1. AN INTRODUCTION TO PROLOG</b>	1
1.1. Data Structures	1
1.2. Operations	11
1.3. Control	24
<b>2. PROLOG AND LOGIC</b>	41
2.1. Introduction	41
2.2. Formulae and Their Interpretations	41
2.3. Formal Reasoning	44
2.4. Resolution and Horn Clauses	45
2.5. Strategy	51
<b>3. METAMORPHOSIS GRAMMARS:     A POWERFUL EXTENSION</b>	59
3.1. Prolog Representation of the Parsing Problem	59
3.2. The Simplest Form of Grammar Rules	67
3.3. Parameters of Non-terminal Symbols	69
3.4. Extensions	73
3.5. Programming Hints	81
<b>4. SIMPLE PROGRAMMING TECHNIQUES</b>	87
4.1. Introduction	87
4.2. Examples of Data Structures	88
4.3. Some Programming Hints	121
4.4. Examples of Program Design	130



<b>5. SUMMARY OF SYNTAX AND BUILT-IN PROCEDURES</b>	143
5.1. Prolog Syntax	143
5.2. Built-in Procedures: General Information	147
5.3. Convenience	149
5.4. Arithmetic	150
5.5. Comparing Integers and Names	151
5.6. Testing Term Equality	152
5.7. Input/Output	153
5.8. Testing Characters	158
5.9. Testing Types	158
5.10. Accessing the Structure of Terms	159
5.11. Accessing Procedures	160
5.12. Control	163
5.13. Debugging	164
5.14. Grammar Processing	165
5.15. Miscellaneous	165
<b>6. PRINCIPLES OF PROLOG IMPLEMENTATION</b>	167
6.1. Introduction	167
6.2. Representation of Terms	168
6.3. Control	178
6.4. Tail Recursion Optimisation	179
6.5. Bibliographic Notes	183
<b>7. TOY: AN EXERCISE IN IMPLEMENTATION</b>	185
7.1. Introduction	185
7.2. General Information	186
7.3. The Toy-Prolog Interpreter	187
7.4. Interpretation of Prolog-10 in Toy-Prolog	201
<b>8. TWO CASE STUDIES</b>	215
8.1. Planning	215
8.2. Prolog and Relational Data Bases	226
<b>9. PROLOG DIALECTS</b>	249
9.1. Prolog I	249
9.2. Prolog II	250
9.3. Micro-Prolog and MPROLOG	253
<b>APPENDICES</b>	255
A.1. Kernel File	256
A.2. "Bootstrapper"	258



Contents	ix
A.3. User Interface and Utilities	263
A.4. Three Useful Programs	284
<b>REFERENCES</b>	293
<b>INDEX</b>	301



---

## PREFACE

---

Prolog is a non-conventional programming language for a wide spectrum of applications, including language processing, data base modelling and implementation, symbolic computing, expert systems, computer-aided design, simulation, software prototyping and planning. A version of Prolog has been chosen as a systems programming language for so-called fifth-generation computers; experiments with systems programming and concurrent programming are in progress.

Prolog, devised by Alain Colmerauer, is a logic programming language. Logic programming is a new discipline which lends a unifying view to many domains of computer science. Prolog can be classified as a descriptive programming language, as opposed to prescriptive (or imperative) languages such as Pascal, C and Ada. In principle, the programmer is only supposed to specify *what* is to be done by his or her program, without bothering with *how* this should be achieved. Robert A. Kowalski has coined the "equation"

$$\text{Algorithm} = \text{Logic} + \text{Control},$$

which emphasizes the distinction between the *what* (logic) and the *how* (control). The programmer need not always specify the control component. In practice, however, Prolog can be treated as a procedural language.

Prolog is not standardized, and it comes in many different flavours. The most widespread dialect of Prolog is Prolog-10, originally implemented by David H. D. Warren for DEC-10 computers. We describe a variant of this dialect, based on an interpreter written in Pascal especially for this book.

The main part of the book is Chapters 1–5. Chapters 1 and 3 are an introduction to Prolog, intended for those who use prescriptive languages in their everyday practice. Both intuitions and the presentation are "practically" biased, but we assume the reader has a certain amount of programming experience and sophistication. Chapter 2 explains Prolog in



terms of logic. It requires no deep knowledge of mathematics and is intended as a counterpoint to Chapter 1, but can be skipped on a first reading. Chapter 4 contains some useful programming techniques and hints. Chapter 5 is a reference manual for the version of Prolog described in this book. In addition, Chapter 8 is a discussion of two rather illuminating applications.

For those who wish to gain more insight into the language and its inner workings, Chapter 6 introduces basic principles of Prolog implementation. An implementation of the dialect described in this book is presented in Chapter 7. We used this implementation to test our examples, including the case studies of Chapter 8.

Chapter 9, written by Janusz S. Bieñ (who also did most of the bibliography), briefly outlines the most characteristic features of several other Prolog dialects.

The diskette enclosed with this book contains source text of all the programs listed in Chapter 8 and in the appendices, and of the Toy-Prolog interpreter discussed in Chapter 7. The interpreter is written in TURBO Pascal. You can use the diskette on any IBM PC compatible computer running MS-DOS 2.10 or 3.10.

The material in this book, supplemented by some additional reading and a programming assignment, can be used for a two-semester course at the level of third-year computer science majors. Re-implementation of or extensions to the interpreter of Chapter 7 might make interesting assignments for a translator-writing course.

While working on this book, we used the computing facilities of the Institute of Informatics, Warsaw University. We would like to thank Paweł Gburzyński and Krzysztof Kimbler, who helped us switch almost painlessly to a different machine when the one we originally used broke down for a protracted period of time. We thank David H. D. Warren for permitting us to include the listings of WARPLAN. We are also grateful to all those who have provided us with logic programming literature for the past 10 years.



---

# 1 AN INTRODUCTION TO PROLOG

---

Prolog is an unconventional language. In particular, its data structures are quite different from those found in other programming languages. As it is difficult to talk about a computation without understanding the sort of data that can be processed, we shall discuss data structures at some length before coming to the question of how to do anything with them. Have patience.

## 1.1. DATA STRUCTURES

### 1.1.1. Constants

**Constants** are the primitive building blocks of data structures. Constants have no structure, so they are often called “atoms.” They represent only themselves—they can be thought of as identical with their names.

In Basic or Fortran, 1951 is a constant. The integer variable **J** is not, because it represents both a memory cell and—in certain contexts—a value. The value is something quite different from the variable itself.

One is accustomed to treating 1951 as a number greater than 1948, but this is because in programming languages constants usually belong to certain types. The usual properties of integer constants (their ordering, ability to be used in arithmetic operations, etc.) are taken for granted by virtue of their belonging to the type **integer**, just as in Pascal blue is a successor of red when one writes

colour = ( red, blue, green )

Such type definitions impose a certain structure on the otherwise undifferentiated universe of individual symbolic constants, each of which has only one attribute: its name.

The interpretation of a constant rests solely with the programmer. 1951 can be the price of a computer, the weight of a truck, the time of day or a year of birth. One can always multiply it by 4, but this seldom makes sense when it represents a car's registration number. The constant blue is less burdened with inadequate interpretations, but one might wish not to have the colours ordered. Constants are the primitives, and collecting them into types should only be done when necessary.

In Prolog, as in other symbolic languages (such as Lisp) there is no need to declare constants or group them into types. One can use them freely, simply by writing down their names.

A legal constant name is one of the following:

- A sequence of digits, possibly prefixed by a minus sign; by convention, such constants are called **integers** (e.g. 0, -7, 1951);
- An **identifier**, which may contain letters, digits and underscores but must begin with a lower case letter (e.g. q, aName, number\_9);
- A symbol which is a nonempty sequence of any of the following characters:

+ - \* / < = > . : ? \$ & @ # \ ~

- Any one of the characters

, or ; or !

- The symbol [] (pronounced "nil");
- A **quoted name**, written according to the Pascal convention for strings: an arbitrary sequence of characters enclosed in apostrophes, an apostrophe being represented by two consecutive apostrophes (e.g. 'Can't do this.' consists of 14 characters).

All of these constants are purely symbolic and have no inherent interpretation. However, some primitive operations in Prolog do treat them in a special way:

- Arithmetic operations interpret integers as representations of integer values (they can also create new integers);
- Comparison operations interpret integers as integer values, and all other constants as representations of the sequences of characters forming their names (these are lexicographically ordered by the underlying collating sequence);
- Input/output operations interpret all symbols as sequences of characters forming their names.



Each occurrence of a constant's description (name) is treated as referring to *the same* constant, but of course we are free to interpret each separately.

### 1.1.2. Compound Objects

An important aspect of the expressive power of a programming language is its ability to *directly* describe various data structures. Of the popular and widely used languages, Pascal is the most powerful in this respect, but it has several shortcomings. (This is *not* a criticism of Pascal: our point of view does not take into account important design objectives such as a safe type mechanism.)

Firstly, type definitions in Pascal are overspecified. It is impossible to program general algorithms which process stacks or trees regardless of the type of their elements. (Records with variants are only a rough approximation to generic data types found in some more recent programming languages.)

Secondly, those data structures which change their form dynamically can only be built with pointers. One must therefore deal with the structures at a very low level: the level of representation rather than the conceptual level at which many other things are done in Pascal. Programs using pointers are error-prone and hard to understand, because operations on such data structures are *encoded* rather than directly *expressed*.

The third shortcoming has a similar effect. Ironically, Pascal types are also *underspecified*, in that there is no way to directly express certain quite natural constraints on the arguments of operations. One cannot say that the function POP can only be applied to a non-empty stack; one can only write a piece of code (hopefully correct) which checks the argument.

It is interesting that these shortcomings are not shared by Prolog data types (or rather by their counterparts, since "type" is not really a Prolog concept). And yet Prolog data structures are very simple. Let us look at the details.

#### FUNCTORS

To describe a compound object, it is not enough to list its components. The ordered pair (19, 24) can be an object of the type

```
rectangle = record
            height, width : integer
          end
```



as well as an object of the type

```
timeofday = record
             hour, minute : integer
end
```

The complete description of a compound object must include a definition of its structure. Structure is defined principally by describing the way in which the object and its components are interrelated. Describing these interrelationships often consists in simply giving a **type name** to an aggregate of components (as in the example above). It is the programmer's responsibility to interpret this name in terms of real-world relations between entities being modelled by the program.

In conventional programming languages, the structure of a compound object is usually described in a declaration associating the object with a type definition. The type definition lists the name of the object-component relationship (type name) and, possibly, additional information about the structure (types) of the components. The object is described by its name (or the name of a pointer). The name's definition is textually remote from its occurrences.

A different approach is taken in Prolog. Here, the type name is an integral part of all the occurrences of the object's description. The notation is very simple: a description of a compound object is the type name followed by a parenthesized sequence of descriptions of its components, separated by commas. We write either

```
rectangle( 19, 24 )
or
timeofday( 19, 24 ).
```

The notation is similar to that used for writing functions in mathematics. Terminology reflects this similarity. The type-name is called a **functor**, and the components are called **arguments**. There is more to it than superficial similarity of two simple syntactic conventions. One can certainly regard a type such as *rectangle* as a function mapping components into compound objects. From this point of view it is not surprising that we can have functors with no arguments: these are simply constants. Sometimes it is also useful to have one-argument functors. For example, the integer 2 can be represented by the object `successor(successor(zero))` (the fact that  $2 > 1 > 0$  is evident from its structure).

From the discussion above, it should be obvious that the important attributes of a functor are both its name and its **arity** (i.e. the number of arguments it takes). In Prolog, we can use both

```
timeofday( 17, 13 )
```



and

timeofday( 1713 )

in the same program. Even if the intended interpretation is the same, these are two different objects: one has two components, and the other has one. There are also two different functors, both named timeofday. Whenever we speak of a functor in a context which gives no indication of its arity, the arity must be given explicitly. The usual notation is to write it after a slash: timeofday/2 or timeofday/1.

The lexical rules for forming functor names are the same for all arities, but integers can only be constants. Thus

123( a, b )

is incorrect, but

'123'( a, b )

is perfectly all right. Also, [] is only a constant.

## OBJECT DESCRIPTIONS

Descriptions of constants and compound objects are referred to as **terms**. Usually, the objects themselves are also called terms: this causes no confusion in practice, but in this chapter we shall try to distinguish between the two meanings.

The arguments of a term are arbitrary terms. For example, one can write a term describing a "record":

customer( name( john, smith ),  
          address( street( north\_ave ), number ( 173 ) ) ).

Of the various functors in this example, the outermost, customer/2, can be said to define the general structure of the term. It is called the **main functor**, or **principal functor**. Similarly, name/2 is the main functor of the first argument.

Here is another example of a common data structure. A list can be defined as either the empty list, or a list constructed of any object (a head) and a list (a tail). A list of the first three letters in the alphabet could then be described by the term

cons( a, cons( b, cons( c, emptylist ) ) ).

The following term would be a description of the two-element list constructed of the above list and a list containing the integer zero:

cons( cons( a, cons( b, cons( c, emptylist ) ) ), cons( 0, emptylist ) )



Even this small example demonstrates that nested parentheses can be difficult to read. Prolog therefore provides syntactic sugar to hide this **standard** or **canonic form** of terms. Instead of writing

```
successor( successor( zero ) )
```

one can choose to use *successor* as a **prefix functor** and write

```
successor successor zero.
```

Alternatively, *successor* can be made a **postfix functor**:

```
zero successor successor.
```

Functors with two arguments can be declared as **infix functors**, e.g.

```
&( a, b )
```

can be written as

```
a & b.
```

The term

```
a & b & c
```

would be ambiguous, so an infix functor is either **left-associative** or **right-associative** (or non-associative, in which case the term is incorrect). If & is right associative, the term's standard form is

```
&( a, &( b, c ) ) ;
```

if & is left-associative, then the term stands for

```
&( &( a, b ), c ).
```

We can use parentheses to stress or override associativity. If & is right-associative, then

```
a & b & c
```

is equivalent to

```
a & ( b & c )
```

but not to

```
( a & b ) & c,
```

which stands for

```
&( &( a, b ), c )
```

regardless of associativity.

To make parentheses even less frequent, prefix, postfix and infix functors are given **priorities**. Functors with lower priority take precedence over those with a higher priority (a Prolog-10 convention, different from that used in other programming languages and mathematics). For example, if the priority of  $*$  is lower than that of  $+$ , then

$$3 * 4 + 5 \quad \text{and} \quad 5 + 3 * 4$$

denote

$$+( *( 3, 4 ), 5 ) \quad \text{and} \quad +( 5, *( 3, 4 ) )$$

We can use parentheses to stress or override priorities, by writing

$$( 3 * 4 ) + 5 \quad \text{or} \quad 3 * ( 4 + 5 ).$$

Prefix, postfix and infix functors are usually referred to by the generic name **operators**. Remember that these are not operators in any conventional sense: they are only a syntactic convenience.

Operator names may not be quoted. If an operator is to be written in standard form or with a different number of arguments, it must be quoted. If  $+$  is an infix functor,

$$a + b, \quad '+'( a, b ) \quad \text{and} \quad '+'( a, b, c )$$

are correct terms, but

$$+ \quad \text{and} \quad +( a, b )$$

are not.

It is also possible to declare **mixed operators**, i.e. functors such as the minus sign, which is both prefix and infix in ordinary arithmetic. Details about declaring prefix, postfix and infix functors can be found in Sections 5.1 and 5.7.3.

For the time being, we shall only use infix functors to write terms representing lists. However, instead of

$$a \text{ cons } b \text{ cons } c \text{ cons emptylist}$$

we shall use a more concise notation, modelled after Lisp. The empty list will be denoted by the constant  $[]$  (pronounced "nil"), and the constructing functor – by the right-associative infix functor  $./2$ . Our two lists are then written as

$$a.b.c.[ ]$$

and

$$( a.b.c.[ ] ).0.[ ]$$



The convention is arbitrary, in that any constant and two-argument functor would do in place of [] and the dot. It is more convenient than others, because these are the symbols expected by several built-in procedures.

(You can write such terms after feeding Prolog with

```
:- op( 800, xfy, '.' ).
```

However, a minor technical difficulty makes it impossible to use the period as a functor when it is immediately followed by a white space character, such as blank, tab or new line. This is a nuisance, and Prolog provides special syntactic sugar for lists: it is somewhat confusing, so we will put it off until Chapter 4.)

## STRINGS

Characters are constants whose names consist of single characters. One can use quoted names for characters which are not correct identifiers (e.g. ' ', '(', '3'; and 'x' is equivalent to x).

**Strings** are lists of characters. One can also write them in double quotes. For example

```
"string" and " """"
```

stand for

```
s.t.r.i.n.g.[ ] and ' '. '''.[ ]
```

(Actually, the convention adopted in this book is different from that of Prolog-10. There, a string denotes a list of ASCII codes and not a list of characters, so "string" stands for

```
115.116.114.105.110.103.[ ].
```

Similarly, in Prolog-10 operations for reading and writing characters deal directly with ASCII codes. We refuse to accept these conventions.)

### 1.1.3. Variables

Objects discussed so far are all, in a sense, constant. Their structure is fixed, we know everything about them and cannot learn anything new. A programming language in which one could specify only such fully defined objects would hardly be interesting. One must be able to use objects whose complete form is defined dynamically during a computation.

In Prolog, the simplest such as-yet-unknown objects are called **variables** (do not confuse them even for a moment with the variables of conventional programming languages!). The term denoting a variable is



called a **variable name** (this is also usually called a variable: as with terms and objects, we shall try to maintain the distinction throughout chapter 1). A variable name is written as an identifier starting with an upper case letter or an underscore (e.g. `Q`, `Number_9`, `_nnn`).

A variable is an object whose structure is totally unknown. As a computation progresses, the variable may become **instantiated**, i.e. a more precise description of the object may be determined. The term embodying this description is called the variable's **instantiation**. An instantiated variable is identical with the object described by its instantiation, so it ceases to be a variable, although the object can still be referred to through the variable's name. (In general, a variable may be instantiated also to another variable—we shall soon see the meaning of this.)

There is also an alternative terminology. One says that a **free** (or **unbound**) variable becomes **bound** to another term and is henceforth indistinguishable from that term (which is called its **binding**). The variable becomes **ground** if its binding contains no variables. This terminology brings to mind the process of binding formal parameters to actual parameters. If the formal parameters were not allowed to change their value (as in pure Lisp, say), the similarity would be very close indeed, except that a binding need not be ground.

Intuitively, Prolog variables are somewhat like the variables used in mathematics. When we say that

$$f(x) = e^x + 3x$$

is a function of one variable, we mean that the equation allows us to determine the function's value for any (one) given argument. The variable denotes a single (albeit arbitrary) substitution and is not in itself an object to which values can be assigned.

You can also regard a Prolog variable as an "invisible" pointer. When not free, the pointer is automatically dereferenced in all contexts, so it is impossible to distinguish it from the referenced object: in particular, it is impossible to exchange the object for something else.

#### 1.1.4. Terms

If one thinks of a type as a set of objects, then a term is also a definition of a type. The term `Variable3` describes the set of all objects, because a variable can be instantiated to anything. On the other hand, one can have a very precise type specification. For example, the term `a.b.c.[]` describes a set containing only one object: the list of length 3, whose first element is `a`, whose second element is `b` and whose third element is `c`. There is a wide range of choices between these extremes.



We describe objects by defining those of their properties which we find interesting in a given context. We do so by using variable names to denote objects (in particular: components of other objects) whose exact form is either unknown or unimportant. Our descriptions thus denote sets of objects satisfying the explicitly formulated properties.

A few examples should make it clear:

1. painting( Painter, 'Saskia' )  
—all 'Saskia's of an unknown artist
2. painting( rembrandt, Picture )  
—all pictures by Rembrandt
3. painting( rembrandt, picture(Title,1646) )  
—all pictures painted by Rembrandt in 1646
4. Head.Tail  
—all non-empty lists
5. One.Two.Three.[ ]  
—all lists of three elements
6. One.Two.13.Tail  
—all lists containing at least three elements, such that the third element is the number 13.

Actually, our comments in examples 4 and 6 are somewhat imprecise, as they reflect an intended interpretation. Since a variable name denotes an arbitrary object, the type Head.Tail contains more than true lists: the object one.two also answers this description. Similarly, the term in example 1 describes objects such as painting(59, 'Saskia'). Term notation does not allow us to express directly our wish to consider only paintings whose first arguments are the names of painters. This is in keeping with the principle that the type of a compound object is defined primarily by the interrelationship between the object and its components, rather than by the types of the components. The restriction is not necessarily a bad thing: we shall see that a procedure popping an element off a stack is most naturally written so that it can handle all stacks, whatever the types of their elements. If one considers it important to restrict the types of components, one can do it easily enough (we shall see how), but only consciously and only when needed.

As a computation progresses, variables in various terms may become instantiated. As a result, more is known about the objects described by these terms. We extend our terminology so that we can talk about **instantiating terms** and terms which are **instantiations of other terms**. For example, f(X).Tail is an instantiation of Head.Tail; and it may, in due course, be further instantiated to a yet more precise description. As we shall see,



such a multi-step approximation to a desired description is very characteristic of Prolog.

We have proposed to regard a term as a type definition, i.e. a description of a class of objects, or alternatively as a description of a single, as yet undefined object. These are two sides of the same coin. A single object whose form is known only in general outline can be thought of as a representative of the class of all objects having that form. A term denotes a set by virtue of denoting any one of its possible instantiations.

A comment on the role of variable names. They are used as handles on the objects they denote. Through a name we can, at any moment, look at what we have actually learned about the shape of the object. For example, no matter what the instantiation of `Head.Tail`, the variable name `Head` denotes the first element of this list. The term `X.X.Tail` is also quite legal in Prolog and denotes a list whose first and second elements are the same object.

Notice that we said "the same object," not "identical objects." It is important to note that different compound objects can **share** components. In general, Prolog terms describe data structures which can be represented as directed acyclic graphs (DAGs). If we use an arrow to denote the relation of "being built of" ( $X \rightarrow Y$  means that  $Y$  is a component of  $X$ ), then Fig. 1.1 illustrates the object denoted by

`one( two( A.B ), three( A.B, C ), B ).`

Sometimes one is not interested in certain objects and needs no name to refer to them. Such terms can be denoted by **anonymous variables**, each of which is written as an underscore. For example

`a._._.b.[ ]`

describes a list of length four whose first and last elements are `a` and `b`. The second and third elements can be any two different objects, or the same object: we don't care.

## 1.2. OPERATIONS

The majority of operations in a Prolog program are calls to procedures defined by the user. Standard operations—addition, comparisons, input/output etc.—are used relatively infrequently. For uniformity, **every** operation is written as if it were a procedure call, and the principal property of all standard operations is only that they need not (and must not) be de-



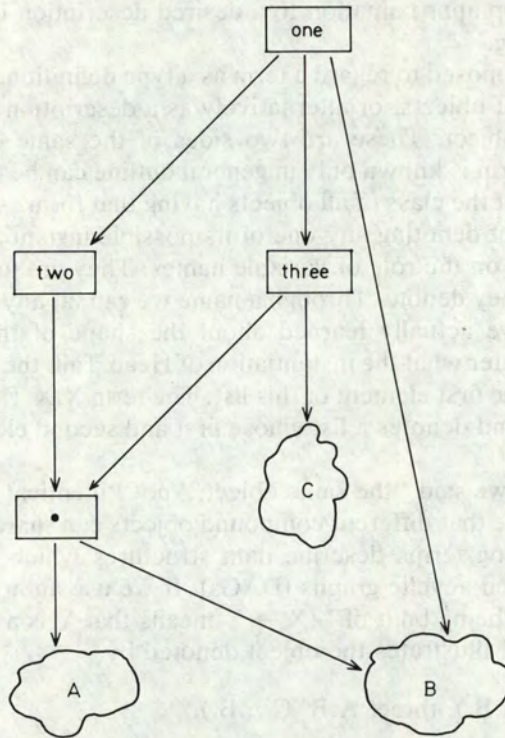


FIG. 1.1 Recurring components of an object.

defined by the user. Standard operations are accordingly called **built-in procedures** (or **system procedures**).

Procedure calls are written according to the usual practice. The procedure name is followed by an optional list of terms—actual parameters, enclosed in parentheses and separated by commas, e.g.

`show( painting(rembrandt,X), etching(rembrandt,X) ).`

The “procedure name” is often called a **predicate symbol** (sometimes shortened to **predicate**). Like functors, a predicate symbol has two attributes: a name and an arity. Two distinct procedures can share the same name, provided one has a different number of parameters than the other.

Procedure calls (and, as we shall see, procedure definitions) have the same syntax as terms. This notational uniformity is useful when programs are dynamically modified (as is normally the case during an interactive session), but it may be confusing for the uninitiated. We shall try to help by reserving the word “argument” for components of terms: procedures will be said to have parameters.



Some versions of Prolog—including those described in this book—carry this uniformity to the point of allowing the user to use prefix, postfix and infix notation for predicate symbols (such symbols are also called “operators”). This is achieved exactly as for functors, but a number of frequently used symbols are usually predeclared to give the language a more conventional flavour. A case in point is the built-in procedure *is*, whose name is written in infix notation. It expects its second parameter to be an “expression”: a term representing the abstract syntax tree of an integer arithmetic expression; the tree is evaluated and the result is returned through the first parameter. The two-argument functors  $+$ ,  $-$ ,  $*$ ,  $/$  and *mod* are predeclared as infix functors with conventional priorities and associativity, so one can write

V is X - 7 - Y \* Z mod ( 2 + X )

to instantiate V to an integer (provided the instantiations of X, Y and Z are integers or “expressions”).

Sequences of procedure calls use commas for separators, for example:

... buy( picture(rembrandt,Title), Price ),  
       NewPrice is Price \* 135/100,  
       sell( picture(rembrandt,Title), NewPrice ),  
       drink( beer ) ...

(We shall enlarge on this in Sections 1.2.2 and 1.3.1.)

We shall need two built-in procedures for our examples:

- nl/0 terminates an output line;
- write/1 outputs a term (variables are written as X1, X2, etc.); for example, if A and B are uninstantiated, then

write( f('an id',g(A,B),7,A) ), nl

writes

f( an id, g( X1, X2 ), 7, X1 ).

More precise descriptions of system procedures can be found in Chapter 5. We shall now see how to define user procedures.

### 1.2.1. The Simplest Form of a Procedure

Try to think of a procedure which computes the head and the tail of a list: we shall call it *carcdr*. What should its specification be like?

Let the list be the first parameter and let the second and third parameters return its head and its tail. Head and tail are defined only for non-



empty lists, so the first parameter's type is described by the term `Head.Tail` (`Q1.Q2` would do just as well, but it is better to use meaningful names). This type specification is most naturally written in the procedure heading, thus

```
carcdr( Head.Tail, ..., ... ) ...
```

If a list is denoted by `Head.Tail`, then `Head` denotes its head and `Tail` denotes its tail. We can therefore write

```
carcdr( Head.Tail, Head, Tail ).
```

The fullstop terminates the specification. It is rather concise, but it contains all the necessary information; the procedure is called `carcdr` and has three parameters; the first parameter must be a non-empty list, the second parameter is to become the head, and the third is to become the tail of this list.

It turns out that what we have written is also the complete definition of this procedure in Prolog. The call

```
carcdr( 1.2.3.[], H, T )
```

instantiates `H` to `1` and `T` to `2.3.[]`. (Recall that `1.2.3.[]` is really `.(1,.(2,.(3,[])))`.)

Actually, our definition is somewhat more general, because—as we have already pointed out—`Head.Tail` need not be a list, as no conditions are imposed on the form of its tail. But this does not matter: there is no misunderstanding about the desired effect of, say

```
carcdr( timeofday( 12, 30 ).any( Object, "at all" ), F, S ).
```

What we have here is a general procedure for getting at the first and second arguments of a term whose main functor is `./2`.

We shall now specify the reverse of `carcdr`: a procedure which returns, via its third parameter, a list constructed of its first and second parameters. We shall call it `cons`.

We do not really mind if the first two parameters are not lists, so there are no restrictions on their types:

```
cons( Object, Another, ... ) ...
```

If `Object` describes an object and `Another` describes an object, then applying the list constructor to the two gives us a third object, whose description is `Object.Another`. This term is sufficient as a specification of the third parameter, so we get

```
cons( Object, Another, Object.Another ).
```



Here, again, we have a complete definition of this procedure. But notice that the order of parameters has no meaning in itself, so we might as well have decided to pass the constructed list through the first parameter:

```
cons( Object.Another, Object, Another ).
```

Variable names have no inherent meaning either, so `cons` is really the same as `carcdr`. Indeed, when we read out the specification of `carcdr`, we cheated a little: "the parameter must be," or "the parameter is to become"—these distinctions were not present in the specification.

While both `carcdr` and `cons` could be so named to reflect their intended use, they are both really a single procedure

```
conscardr( Head.Tail, Head, Tail ).
```

This is not very surprising, as `cons` is the reverse of the coin of which `carcdr` is the face.

Now the call

```
conscardr( 1.2.3.[], H, T )
```

instantiates `H` to `1` and `T` to `2.3.[]` and the call

```
conscardr( L, a, b.[] )
```

instantiates `L` to `a.b.[]`. But how is it done? We shall come to that, as soon as we have cleared up a point of syntax.

### 1.2.2. Directives

In versions of Prolog deriving from Prolog-10, the syntax of a simple procedure definition such as our `conscardr` example need not necessarily differ from that of a procedure call. The meaning is defined by context.

Such Prolog systems function in two modes: the command mode and the definition mode. Command mode is the default.

In command mode, the system reads and executes **directives**. The directives are read in from the user's terminal or from a file. Each directive is terminated by a fullstop (the character `.`, immediately followed by a white space character, including newline), and is either a **query** or a **command**. A query is a procedure call or a sequence of procedure calls separated by commas. Roughly, its execution consists of executing its call and printing the resulting variable instantiations (see the end of Section 1.2.3 for a more precise description). For example, if `conscardr` has been



defined, then after reading the query

```
conscardr( 1.2.3.[], H, T ).
```

the system writes

```
H = 1
```

```
T = 2.3.[]
```

(The actual printout might be in the special syntax used for lists; see Section 4.2.1.)

A command has the form of a query prefixed by the symbol `:-`. Its calls are executed but the variable instantiations are not written out automatically. To get the same printout with a command, one would write

```
:- conscardr( 1.2.3.[], H, T ),
   write('H = '), write( H ), nl,
   write('T = '), write( T ), nl.
```

The terminology is somewhat fluid: directives are often called **goal statements**, while queries and commands are not always recognized under those names. (Sometimes there are also slight syntactic differences. We try to follow the original definition of Prolog-10, but in this book the standard is set by the version of Prolog described in Chapter 7.)

Definition mode is entered upon executing the system procedure `consult/1` or `reconsult/1`. (The argument is the name of the file from which procedure definitions are to be read; *user* is the name of the user's terminal. The details are in Section 5.11.) In this mode, the system accepts procedure definitions, which are also terminated by fullstops. Our definition of `conscardr` is an example, but see Section 1.3.1 for the complete syntax. Commands are allowed and properly executed in this mode, but queries are not. Definition mode is exited when the system encounters the definition

```
end.
```

A note about comments in Prolog. A comment starts with a `%` character (not contained in a string or quoted name) and extends till the end of line. Be careful not to place a comment immediately after a dot that terminates a clause: a fullstop is required.

As a point of interest, all directives and the basic building blocks of procedures (called clauses—we will describe them in due time) are simply single terms. Standard operator declarations include the infix functor `,` (comma) and the prefix functor `:-`, so the directive

```
:- p( 2, X ), write( X ), nl.
```

is really the term

```
':-'(' ','( p( 2, X ), ','( write( X ), nl ) ) ).
```

This data structure is interpreted as a directive, so you need not worry about these things unless you are an advanced Prolog hacker.

There is one important point, though probably you will find it obvious. The actual parameters of procedure calls are the current instantiations of terms directly written in the call. Thus

```
:- conscarcdr( a.b.[] , H , T ), conscarcdr( L , H , T ), write( L ), nl.
```

will print out

```
a.b.[]
```

### 1.2.3. Unification

Since we succeeded in packing the whole definition of `conscardr` into its heading—the part specifying its name and formal parameters—we can expect that its execution boils down to applying a sufficiently powerful and general parameter-passing mechanism. This mechanism is implemented by a term-matching operation called **unification**.

We will describe this operation by a pidgin—Pascal algorithm. The function `UNIFY` is applied in turn to each formal and actual parameter pair. If it returns true for all such pairs of terms, we say that unification is **successful** (or **succeeds**); otherwise unification **fails**. Unification fails when the terms describing the parameters do not match. In a very general sense this means that the types of actual parameters are incompatible with those of the formal parameters.

```
function UNIFY ( var Actual, Formal : term ) : boolean;
var success : boolean;
begin success:= true;
  if Formal is a variable then
    Formal is instantiated to Actual
  else
    if Actual is a variable then
      Actual is instantiated to Formal
    else
      if the main functors of Formal and Actual have
        different names or arities then success:= false
      else
        while success and unmatched arguments remain do
          success:= UNIFY( next argument of Actual,
                           next argument of Formal );
    UNIFY:= success
end;
```



Notice that if we treat both the call and the procedure heading as terms, then the process of matching successive pairs of parameters is subsumed by the loop in UNIFY. We extend our terminology accordingly, and say that—like a pair of terms—a call and a procedure heading do or do not **match**. Alternatively, we say that they do or do not **unify** (are or are not **unifiable**). The algorithm **unifies** matching terms. Unified terms are indistinguishable, so they describe the same object.

If both Formal and Actual describe variables, then unification **binds them together**. Variables which are bound together also represent the same object: both their names refer to the same variable. (It is pointless to ask whether the formal becomes an instantiation of the actual or the other way round. Our algorithm implements the latter case, but this is not observable from the outside. You can envisage a set of bound-together variables as a chain of invisible pointers.)

Time for a very detailed analysis of a simple example: the procedure

$p(A, b(c, A))$

called with the query

$p(X, b(X, Y))$ .

Figure 1.2 shows the situation immediately before unification. The horizontal line separates objects local to the directive and objects local to

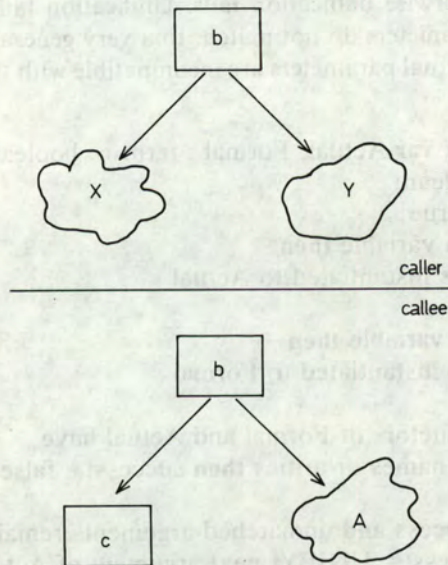


FIG. 1.2 Unification: before matching.

the procedure. Note that objects—including variables—are accessible both directly through their names and as components of other objects.

The first pair of parameters is matched by binding  $X$  and  $A$  together. The variables will behave as if they had merged into a simple object (somewhat like two drops of water). This object is accessible under two different names (Fig. 1.3).

The second pair of parameters is unified in two phases. First,  $c$  becomes the instantiation of the “amalgamated” variables  $X$  and  $A$ . They cease to exist as variables, but  $c$  is now also accessible through the name  $A$  inside the procedure and through the name  $X$  outside (Fig. 1.4).

In the second phase  $Y$  is instantiated to the instantiation of  $A$ . The object  $c$  is now accessible as  $c$ ,  $A$ ,  $X$  and  $Y$  (Fig. 1.5).

The procedure  $p$  now terminates, but its local object  $c$  remains, being accessible from the outside as  $X$  or  $Y$ . The instantiation of  $b(X, Y)$  is  $b(c, c)$  (Fig. 1.6).

It is convenient to use a special notation for showing the effects of unification. We shall write

$$A \leftarrow B.[]$$

instead of “ $A$  is instantiated to  $B.[]$ ”; and

$$A \leftrightarrow B$$

instead of “ $A$  and  $B$  are bound together.”

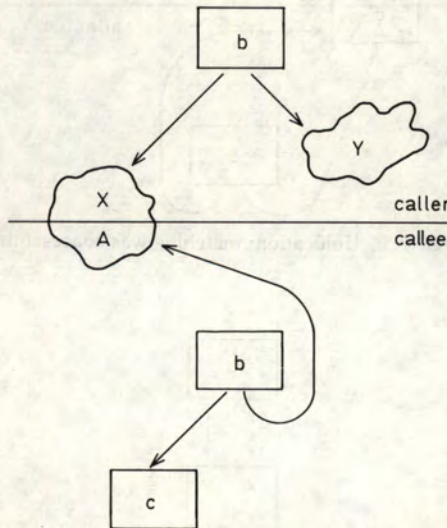


FIG. 1.3 Unification: the first pair of parameters is matched.



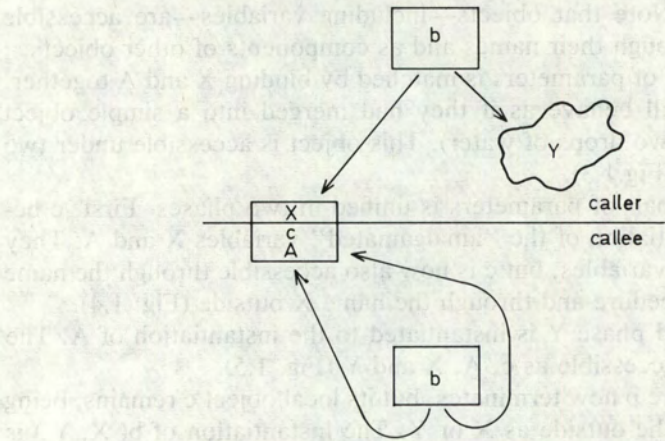


FIG. 1.4 Unification: the first pair of b's arguments is matched.

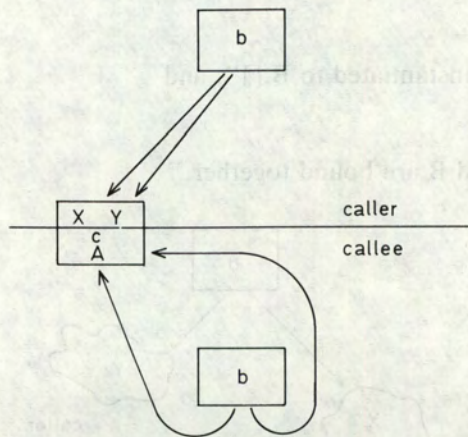
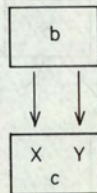


FIG. 1.5 Unification: matching was successful.



**FIG. 1.6 Unification: the callee is terminated.**

Here are some example calls to our procedure

`conscardr( Head.Tail, Head, Tail ).`

1. `conscardr( element.[], Car, Cdr )`  
 $\text{Head} \leftarrow \text{element}, \text{Tail} \leftarrow [],$   
 $\text{Car} \leftarrow \text{element}, \text{Cdr} \leftarrow [].$
2. `conscardr( L, one.two.[], 3.4.5.[] )`  
 $\text{L} \leftarrow \text{Head.Tail}, \text{Head} \leftarrow \text{one.two.[]},$   
 $\text{Tail} \leftarrow 3.4.5.[].$

Hence the instantiation of L is

$(\text{one.two.[]}).3.4.5.[]$

3. `conscardr( A.B, 2, 2.[] )`  
 $A \leftrightarrow \text{Head}, B \leftrightarrow \text{Tail},$   
 $\text{Head} \leftarrow 2 \text{ (and hence also } A \leftarrow 2 \text{),}$   
 $\text{Tail} \leftarrow 2.[] \text{ (and hence also } B \leftarrow 2.[] \text{).}$

The instantiation of A.B is now  
 $2.2.[].$

4. `conscardr( A.B.C, 10, [] )`  
 $A \leftrightarrow \text{Head}, \text{Tail} \leftarrow B.C,$   
 $\text{Head} \leftarrow 10, \text{failure}.$

Unification fails. This is not surprising, as the first actual parameter describes a list of at least two elements, while the list constructed of the second and third actual parameters would have only one element.

We shall wind this up with four general remarks.

First, we want to stress that the unification algorithm treats actual and formal parameters absolutely symmetrically. This results in a very characteristic property of Prolog: there is no difference between formal parameters used to bring information into a procedure and those used to carry information out of a procedure. The direction of information flow changes from call to call, as in examples 1 and 2 above. We can even make a parameter serve both for input and for output. An example is the call

`conscardr( A.2.[], 1, B ).`

Here,  $\text{Head} \leftarrow A$  and  $\text{Tail} \leftarrow 2.[];$   
 then  $\text{Head} \leftarrow 1$  ( and therefore  $A \leftarrow 1$  )  
 and  $B \leftarrow 2.[]$  .

The result is that the first formal parameter was used both for obtaining information (that  $2.[]$ ) and for yielding information (that 1).

This multi-way functioning of procedure parameters sometimes makes it possible to use procedures in unexpected ways. Whenever we



shall say that a procedure does this and this, we shall not worry about what it does after an “unreasonable” call. But you might find thinking about these things a useful exercise.

The second remark: effects of unification such as merging two variables are quite consistent with the interpretation of terms as descriptions of types. We shall illustrate this with a very simple example. The following two procedures accept only three-field records whose neighbouring fields are identical:

```
first2( record( Field12, Field12, Field3 ) ).
last2( record( Field1, Field23, Field23 ) ).
```

Each of these procedures can be thought of as imposing a constraint on a description of the record type. The constraints are not mutually inconsistent, so the directive

```
:- first2( record( F1,F2,F3 ) ),
   last2( record( F1,F2,F3 ) ),
   write( record( F1,F2,F3 ) ), nl.
```

writes out the description of a record whose three fields are identical:

```
record( X1, X1, X1 ).
```

Similarly, the query

```
first2( record( F1,F2,field ) ), last2( record( F1,F2,field ) ).
```

is answered with

```
F1 = field
F2 = field
```

Third, the convention that only the most interesting aspects of an object are captured in a type description turned out to be quite useful. Example 3 would not have worked if the type of the first formal parameter had specified that the tail must be a proper list. The term *A.B* would not have been accepted.

The fourth, and last, remark. If you follow the unification algorithm carefully, you will notice that it can create cyclic data structures. For example, if the procedure

```
same( X, X ).
```

is invoked with

```
same( f( V ), V ), write( V ), nl.
```



then we are in trouble. First,  $X \leftarrow f(V)$ ; then  $V \leftarrow X$ , that is to say  $V \leftarrow f(V)$ . As a result,  $f$  becomes its own component and the printout will be potentially infinite:

$$f(f(f(f(f(f(f(f(f(f(f(f( \dots$$

Such cyclic structures can also cause trouble during unification. If we write

same( f( V ), V ), same( f( W ), W ), same( V, W ),...

then the unification algorithm will not terminate for the third procedure call (this will probably manifest itself as recursion stack overflow):  $f$  matches  $f$ , their first arguments are both  $f$ , and *their* first arguments are both  $f$ , and so on.

All this could be avoided if a variable were not unifiable with a term in which that variable occurs. The unification algorithm is borrowed from automatic theorem proving (see Chapter 2). The original algorithm contains this **occur check**, but most versions of Prolog do not, as it considerably increases the algorithm's time complexity. Fortunately, cyclic structures seldom occur in practice, and one learns to live with the knowledge that terms are not always DAGs if one blunders badly. One version of Prolog (Prolog II; see Section 9.2) is built to take advantage of cyclic data structures. They are called **infinite trees** and are treated as bona fide representations of graphs arising in the real world. If one is careful, one can use such structures even in more conventional Prolog systems: an example is the calltree program listed in Appendix A.4.

#### 1.2.4. Clauses

If we want a procedure which computes the fourth element of a list, we can write

fourth( \_...\_.E4.\_, E4 ).

But this method is useless if we want the n-th (or even the hundredth) element.

After parameters are passed, a procedure can—just as in other languages—execute a sequence of operations. For example, a procedure which prints the fourth element of a list would be:

```
fourth( _._._.E4._ ) :- write( E4 ), nl.
```

Its body is a sequence of calls, separated by commas and prefixed by a :-.  
As you see, a command is like a procedure without a heading.



A procedure heading, possibly followed by a body, is called a **clause**. We shall now see how to use clauses for less trivial tasks.

### 1.3. CONTROL

#### 1.3.1. The General Form of a Procedure

What happens when unification fails?

Part of the answer is that a procedure can consist of a number of clauses. All these clauses must have headings with the same predicate symbol, but the parameter specifications may differ. When unification of a call with the first clause's heading is successful, the first clause executes its body (if any). When unification fails, its effects are **undone**: all variables which were instantiated by the attempt at unification are restored to their original, unbound state. The call is then matched against the heading of the second clause. If this is successful, the second clause is executed; otherwise the third clause is attempted and so on. To execute a procedure is thus to execute the first of its clauses whose head matches the call (but see the next section for a refinement of this statement). Roughly, the matching clause contains code for that particular combination of parameter types.

An elementary example is provided by an extended version of the procedure `carcdr` of Section 1.2.1.

```
carcdr( Head.Tail, Head, Tail ).
carcdr( [], _, _ ) :- write( 'can't crack empty list' ), nl.
```

Here is a somewhat less trivial example, an immortal classic of introductory Prolog courses. It is a procedure which appends a list at the end of another list:

```
append( Hd.Tl, List, Hd.TlAndList ) :-
    append( Tl, List, TlAndList ).
append( [], List, List ).
```

All terms written in a clause are **local to that clause**. Both occurrences of `List` in the first clause refer to the same variable, which has nothing to do with the variable named `List` in the second clause. The second clause might as well have been

```
append( [], Q14, Q14 ).
```

As in other programming languages with recursion, activation of a clause is accompanied by creation of new instances of all its local objects.



The terms appearing in the clause describe these instances. Before an attempt to unify a call with a clause heading can be made, a new instance of the clause is created. When unification fails, the instance is destroyed.

Armed with this knowledge, we can now watch the effects of calling `append` in the query

```
append( a.b.[], c.d.[], Result ).
```

For clarity, we shall use  $X'$ ,  $X''$ , etc., to denote different instances of a variable named  $X$ .

The original call will successfully activate the first clause, after the following instantiations:

```
Hd' ← a, Tl' ← b.[], List' ← c.d.[],
Result ← a.TlAndList' (because Hd' is now a).
```

This clause will execute the call

```
append( b.[], c.d.[], TlAndList' ),
```

activating a second instance of the first clause:

```
Hd'' ← b, Tl'' ← [], List'' ← c.d.[],
TlAndList' ← b.TlAndList'' .
```

In this instance, the body is

```
append( [], c.d.[], TlAndList'' ) .
```

This call does not match the first clause's heading, so the second clause is used:

```
List''' ← c.d.[], TlAndList'' ← c.d.[] .
```

The third instance of `append` has no calls to execute, so it returns to the second instance. The second instance is done with its body, so it returns to the first instance, which also terminates. The variable `Result` was instantiated to `a.TlAndList'`, and `TlAndList'` to `b.TlAndList''`, and `TlAndList''` to `c.d.[]`. Therefore, the query can be answered with

```
Result = a.b.c.d.[]
```

It is sometimes useful to represent the state of a computation by the sequence of calls which must be executed. The sequence is often called the current **resolvent** (see Section 2.4). If we use our procedure in the directive

```
:- append( a.b.[], c.d.[], Result ), write( Result ), nl.
```

then the successive resolvents are as follows:



1. `append(a.b.[],c.d.[],Result), write(Result), nl.`
2. `append(b.[],c.d.[],TlAndList'), write(a.TlAndList'), nl.`
3. `append([],c.d.[],TlAndList''), write(a.b.TlAndList''), nl.`
4. `write(a.b.c.d.[]), nl.`
5. `nl.`

When no calls remain, the directive is terminated.

Here is the procedure to find the  $n$ -th element of a list. Its first parameter is  $n$  and the second a list. The third parameter returns the  $n$ -th element of the list; if the element does not exist, the constant `?` is returned and an error message is printed. It is assumed that the first parameter is not negative (we will learn how to check this in the next section). The procedure is

```

nth( 0, _, ? ) :- write( 'nth( 0,... )??' ), nl.
nth( N, [], ? ) :- write( 'nth( ..,too short,... )??' ), nl.
nth( 1, E1_., E1 ).
nth( N, _:Tail, El ) :- M is N - 1, nth( M, Tail, El ).

```

Do trace its execution for a few examples.

### 1.3.2. Backtracking

But what if a call matches none of the clause headings? An example is the call

```
conscardr( notalist, something, other )
```

This suggests the answer. If none of the clauses fits the call, then evidently the call is wrong: its set of actual parameters does not conform to any of the type specifications describing parameters acceptable to the procedure.

As in other modern programming languages, such an erroneous call does not abnormally terminate a program's execution but activates an error-handling mechanism. In contrast to other languages, however, the error is not necessarily handled by an active procedure present on the activation stack. Prolog uses a more general method and takes into account even those procedures which returned to their caller after successful termination. Procedure instances are looked at, one by one, in reverse order of their activation. The nearest such procedure instance—call it  $p$ —which contains as-yet-unactivated clauses matching its call is assumed to be able to handle the situation. The computation is **undone**: its state is made to appear as if the heading of  $p$ 's most recently activated clause did **not** match its call, and  $p$  is given a chance to execute other clauses.



This process is called **backtracking**, and a call which does not match any clause heading is said to **fail**. Backtracking closely resembles our behaviour in systematically searching for a solution to a problem. If we end up in a blind alley, we get back to the nearest point at which we could have applied another approach, and apply it. If no approach seems to be working at that point, we return to the previous point in which we apparently made a wrong choice, and so on.

In implementation terms, each time a selected clause is not the last in its procedure, a record is pushed onto a special stack of **fail points** (also called **choice points**). The record contains all information necessary to restore the state of the computation. When a procedure fails, the topmost fail point is popped off the stack, the state described by it is restored and the computation proceeds with the next clause.

It is important to note that not all effects of a computation are obliterated on backtracking. Some system procedures do things which cannot be undone, such as writing information on a terminal screen. We say that these procedures have **side-effects**.

Using our description of backtracking, try to follow the execution of procedure `p` in the following example:

```
p :- q( X ), write( trying( X ) ), nl,
    female( X ), write( ok ), nl.
p :- write( 'Sorry!' ), nl.

q( X ) :- writer( X ).
q( ? ) :- write( 'No more writers.' ), nl.

writer( hesse ).
writer( mann ).
writer( grass ).

female( austen ).
female( sand ).
```

You should get the following printout:

```
trying( hesse )
trying( mann )
trying( grass )
No more writers.
trying( ? )
Sorry!
```

You may have noticed from this example that error handling is somewhat inadequate as a metaphor for backtracking. This is the subject of the next section.



### 1.3.3. How to Use Backtracking

When a procedure instance is backtracked to, it behaves as if its most recently activated clause did not match its call. We can therefore use backtracking to implement extended type checking.

Recall from Section 1.1.4 that we found it impossible to write terms which could describe properties such as “the object is a painter’s name” or “the tail is a properly constructed list.” In other words, while rather powerful in certain respects, this kind of type specification is weak in others. This can be remedied by using procedures which do additional type checking and either fail or successfully terminate, depending on the outcome. If we want procedure *q* to accept only properly constructed representations of paintings, we can write

```
q( painting( Painter, Name ) ):-
    ispainter( Painter ), process( painting( Painter,Name ) ).

ispainter( rembrandt ).
ispainter( velasquez ).
.....
```

Here, the role of *ispainter* is similar to the declaration of an enumeration type in Pascal.

Prolog has several built-in procedures which can be used to check properties of objects. For example, one can check whether the object denoted by *Something* is an integer, by seeing whether the call

```
integer( Something )
```

succeeds or fails.

A number of built-in procedures implement comparison operations. Like *is*, procedures for comparing integer values “evaluate” terms resembling conventional arithmetic expressions. The procedures are *<*, *=<*, *:=* (equality), *=\=* (inequality), *>=* and *>*. Their names are predeclared as infix predicate symbols. For example the call

```
7 * 2 + 5 := 1 + 3 * 6
```

will be successful. There are also procedures comparing non-integer constants according to their lexicographic ordering: *@<*, *@=<*, *@>=*, *@>*. For example,

```
alpha @> beta
```

is a failing call.

Equality of constants can be determined by means of the procedure *=*



(= is predeclared as infix). The procedure is most easily expressed in Prolog

```
X = X.
```

It can be used for any two terms, but of course it does more than checking equality. It may cause its parameters to *become* equal, as it attempts to unify them. For example,

```
a( b, X ) = a( Y, c )
```

will succeed after instantiating

```
X ← c, Y ← b .
```

Note that  $7=7$  succeeds, but  $5+2=2+5$  fails, as these are different terms.

Remember that all these procedures do not yield a Boolean result: they only succeed or fail.

When we are interested in the structure of a compound object, we can use a recursive procedure which does nothing but “accepting” the object. Here is a version of *carcdr* which works only for true lists and fails for objects such as *a.b.c* (but not for objects with variable tails, which match []).

```
carcdr( Head.Tail, Head, Tail ) :- islist( Tail ).
```

```
islist( [] ).
```

```
islist( _..L ) :- islist( L ).
```

Such type checking can be quite general. For example, we can process objects differently according to whether they are or are not members of a set represented by a list:

```
process( Obj, Set ) :- member( Obj, Set ),
                        yes_action( Obj ).
```

```
process( Obj, _ ) :- no_action( Obj ).
```

```
member( El, El.Tail ).
```

```
member( El, _..Tail ) :- member( El, Tail ).
```

Try to trace the execution of *process* for a couple of simple calls, and notice how *member* is called with successively shorter tails of the list, until it finds a tail whose head is unifiable with the first parameter.

The fact that the first clause of *member* expresses unifiability rather than equality has very interesting consequences. The most obvious is that the procedure can be used to retrieve information from a dictionary represented by a list. The call



```
member( phone( krull, Number ),
         phone( mann,11 ).phone( hesse,5 ).phone( krull,11 ).[] )
```

instantiates Number to 11.

When this information-retrieving effect is coupled with backtracking, the result is rather striking. Consider the procedure

```
intersect( L1, L2 ) :- member( E, L1 ), member( E, L2 ).
```

When given two sets represented by lists, the procedure terminates successfully if the sets intersect and fails if they are disjoint. Here is a trace of what happens when we call it with

```
intersect( a.b.c.d.[] , c.d.[] ), write( ok ), nl.
```

1. intersect(a.b.c.d.[],c.d.[]), write(ok), nl.

(this activates the procedure:

```
L1 ← a.b.c.d.[] , L2 ← c.d.[]
```

2. member(E,a.b.c.d.[]), member(E,c.d.[]), write(ok), nl.

(activates the first clause of member:

```
E ↔ E1', E1' ← a )
```

3. member(a,c.d.[]), write(ok), nl.

(only the second clause matches the call)

4. member(a,d.[]), write(ok), nl.

(only the second clause matches the call)

5. member(a,[]), write(ok), nl.

(the call to member fails, nearest "handler" is in the procedure activated in step 2, so we backtrack to that situation)

6. member(E,a.b.c.d.[]), member(E,c.d.[]), write(ok), nl.

(the second clause now:

```
E ↔ E1', Tail' ← b.c.d.[] )
```

7. member(E,b.c.d.[]), member(E,c.d.[]), write(ok), nl.

(the first clause:

```
E ↔ E1'', E1'' ← b )
```

8. member(b,c.d.[]), write(ok), nl.

9. member(b,d.[]), write(ok), nl.

10. member(b,[]), write(ok), nl.

(failure, backtracking to step 7)

11. member(E,b.c.d.[]), member(E,c.d.[]), write(ok), nl.

(the second clause now:

```
E ↔ E1''', Tail''' ← c.d.[] )
```

12. member(E,c.d.[]),member(E,c.d.[]), write(ok), nl.

(the first clause:

```
E ↔ E1'''' , E1'''' ← c )
```



13. `member(c,c.d.[]), write(ok), nl.`  
(the first clause)
  14. `write(ok), nl.`
  15. `nl.`
- Success.

Notice how the first call to `member` in `intersect` is used as a “back-track driven” **generator** of successive elements on the list. A terminated procedure can be reactivated if the effects of its execution prove unsatisfactory. It can return several results—or behave in several ways—and its final effect is determined not only by its actual parameters, but also by what happens to the computation later on. It is this multiplicity of possible behaviours that we have in mind when we say that, in general, a Prolog procedure is **nondeterministic**. (This does not mean that its behaviour cannot be predicted to the smallest detail.)

If one wants to see all the results produced by a nondeterministic procedure, one can force Prolog to backtrack by calling an undefined procedure (the call will fail, because there is no matching clause). It is customary to use the name *fail*, both for readability and because Prolog makes it impossible to declare a procedure with this name. To print the elements of a list, one can write

```
:- member( E, a.b.c.[] ), write( E ), nl, fail.
```

Alternatively, one can use a query. After answering a query, the system accepts a single printing character from the terminal. If the character is a semicolon, it backtracks; otherwise it terminates the query. When all the possibilities are exhausted, the word *no* is printed and the system reads another directive. For example,

```
user: female( W ).
system: W = austen
user: ;
system: W = sand
user: ;
system: no
```

If a successful query contains no non-anonymous variables (i.e. no instantiations to show), the answer is *yes*.

Our *intersect* example does more than check for common elements. If the elements are not ground, the sets are modified. For example,

```
intersect( one.X.three.[] , 1.Y.[] )
```



succeeds after binding *Y* to one. This has a natural explanation. Since *Y* is unknown, we cannot say that the sets do not intersect, but by binding *Y* we ensure that the computation will fail if the supposition that *Y* is one will turn out to be unacceptable. We shall then assume that *X* is 1, that *X* and *Y* are the same object, etc., etc.

### 1.3.4. Static Interpretation of Procedures

Detailed simulation of a program is not a very attractive way of learning its meaning. We insisted on doing it to help you understand what happens inside the computer and to introduce techniques which can sometimes be useful for debugging, when things are not happening the way they should. But it is often quite clear what should happen, as many Prolog procedures can be read without giving a thought to details of execution.

A clause which has no body is called a **unit clause**. It is a direct definition of a relation between its parameters. The clause

```
phone( hermann, 5 ).
```

says that hermann and 5 are in the relation *phone*. Other clauses can extend the relation to other objects:

```
phone( mann, 11 ).
```

```
phone( hesse, 5 ).
```

```
phone( krull, 11 ).
```

Unary relations can be thought of as expressing properties of objects:

```
red( herring ).
```

```
red( square ).
```

Nullary relations can denote general facts:

```
tired.
```

```
debugging.
```

A look at the clauses of *phone* tells that the call

```
phone( siddhartha, N )
```

will fail, and the call

```
phone( Who, 5 )
```

will nondeterministically produce hermann and (after a failure) hesse. Note that the calls can be read as “establish whether the actual parame-

ters are in the relation phone, i.e. succeed if they are in the relation, or instantiate them so that they will be in the relation and succeed, or fail."

Somewhat less trivially, the unit clause

`conscarcdr( Head.Tail, Head, Tail ).`

can be used to establish whether the first parameter is a list formed of the second and third parameters. It is self-evident that

1. `conscarcdr( a, b, c )` fails, because the objects are certainly not in the relation;
2. `conscarcdr( A.2.B, 1, C.[ ] )` succeeds, because there does exist a list of at least two elements whose second element is 2, such that its head is 1 and its tail is a one-element list—the list is `1.2.[ ]` and the tail is `2.[ ]`;
3. `conscarcdr( A, B, B.[ ] )` succeeds, because there do exist objects A and B such that A is a list constructed of B and B.[ ]—A is `B.B.[ ]` and B can be any object.

Nonunit clauses are indirect definitions of relations. Thus

`append( [ ], L, L ).`

`append( H.T, L, H.TL ) :- append( T, L, TL ).`

can be read as

"L is L appended to [ ]," and

"H.TL is L appended to H.T if TL is L appended to T."

It is usually convenient to flavour this a little with the intended meaning, as in

"a list L appended to an empty list is L itself," and

"a list L appended to a non-empty list H.T is formed of the head of that list, H, and the result of appending L to its tail, T."

And, most spectacularly,

`intersect( L1, L2 ) :- member( E,L1 ), member( E,L2 ).`

reads:

"L1 and L2 intersect if an object E is a member of L1 and a member of L2",

in other words

"two lists intersect if they have a common member."

You will find more about this in Chapter 2. But note here that this interpretation does not fully explain procedures such as *process* of Section 1.3.3—this is further discussed in Section 4.3.1.



### 1.3.5. The Order of Calls and Clauses

In practice, static interpretation is not always sufficient to explain a program's behaviour. It cannot account for the order of calls in a clause and the order of clauses in a procedure, because "x and y" means the same as "y and x." Yet this order is important, for three principal reasons.

The first reason is that some procedures, such as `write` and `nl`, have side-effects, i.e. their results are not only variable instantiations. The order in which several things are written has an obvious effect on the form of the printout.

Another important reason is efficiency. Here is a famous example (Kowalski 1974) of a naive naive sort:

```
sort( List, Sorted ) :- permute( List, Sorted ),
                        ordered( Sorted ).
```

The procedure generates successive permutations of a list until it finds one that is ordered. If *permute* and *ordered* can be used both to check their parameters and as generators, then this could also be expressed as

```
sort( List, Sorted ) :- ordered( Sorted ),
                        permute( List, Sorted ).
```

Here, successive ordered lists are generated until a permutation of the first parameter is found. Both procedures express the same definition of a sorted list, but while the first is only very costly, the second is absolutely useless.

A third reason is that all computations should be finite. We will illustrate this point with the procedure *append*, which can be written either as

```
append( H.T, L, H.TL ) :- append( T, L, TL ).
append( [], L, L ).
```

or, apparently equivalently, as

```
append( [], L, L ).
append( H.T, L, H.TL ) :- append( T, L, TL ).
```

Both versions are equivalent when *append* is used for appending. But note that its precise reading from section 1.3.4 allows for other uses. For example the second clause, "H.TL is L appended to H.T if TL is L appended to T," defines H.TL in terms of H.T and L, but also H.T and L in terms of H.TL. Indeed, *append* is often used for splitting a list. If one executes



```

:- append( Front, End, a.b.c.[ ] ),
   write( Front ), write( ' & ' ),
   write( End ), nl, fail.

```

then the first version of *append* will produce (after successive failures)

```

a.b.c.[ ] & [ ]
a.b.[ ] & c.[ ]
a.[ ] & b.c.[ ]
[ ] & a.b.c.[ ]

```

and the second version

```

[ ] & a.b.c.[ ]
a.[ ] & b.c.[ ]
a.b.[ ] & c.[ ]
a.b.c.[ ] & [ ].

```

This difference is not very important. But when we write

```
append( L1, a.[ ], L3 )
```

we expect that *append* will succeed, after instantiating the terms so that *L3* is *a.[ ]* appended to *L1*. The second version does just this:  $L1 \leftarrow [ ]$  and  $L3 \leftarrow a.[ ]$ ; then, if we backtrack,  $L1 \leftarrow X1.[ ]$  and  $L3 \leftarrow X1.a.[ ]$ ; then, if we backtrack again,  $L1 \leftarrow X1.X2.[ ]$  and  $L3 \leftarrow X1.X2.a.[ ]$ ; and so on—there are infinitely many such solutions.

The first procedure, however, first looks for the *last* solution in this infinite set, and this causes endless recursion.

Nevertheless, with careful programming, considerations of this sort are needed only to obtain refinements of the general meaning of procedures given by their static interpretation. Moreover, the order of calls and clauses is usually a local thing, seldom requiring looking beyond a single procedure.

### 1.3.6. The Cut

We shall now pass on to so-called **extralogical features** of Prolog. These are simple and powerful mechanisms which play a large part in making Prolog a practical programming language, but cannot be understood in terms of static interpretation, as outlined in Section 1.3.4.

Since we have generators, we must be able to stop them. Suppose that we have two methods for finding the solution of a problem described in terms of two sets. Assume one of these methods is significantly cheaper



than the other, but a necessary—though not sufficient!—condition for its applicability is that the problem-defining sets intersect. We might write something like

```
try( Set1, Set2, Solution ) :-
    intersect( Set1, Set2 ),
    method1( Set1, Set2, Solution ).
try( Set1, Set2, Solution ) :-
    method2( Set1, Set2, Solution ).
```

Now if method1 fails, we want to try method2. But if the sets are large and have many elements in common, we are effectively stopped by a generator. Backtracking from method1 will cause *intersect* to find another way of showing that the sets do indeed intersect: this changes nothing, so method1 will be attempted again and again until *intersect* enumerates all the elements in the intersection of Set1 and Set2. In terms of processing time, this might be a disaster. And note that we are lucky: the generator is not infinite.

To help in such cases, Prolog provides a commit operation, written as ! and called the **cut** procedure (old Prolog hands tend to call it the **slash**, after the character /, which was its name in the original Marseilles Prolog). When procedure p executes a cut, everything that was done by p up to that moment—including its choice of current clause—is taken as fixed and not to be reconsidered on backtracking. In implementation terms, ! cuts away the top section of the fail point stack, leaving only fail points created before p was called.

Our problem can be solved by modifying *intersect*:

```
intersect( L1, L2 ) :- member( E, L1 ),
                      member( E, L2 ), !.
```

The cut kills the generator of elements from L1.

A more involved example might be useful in clearing up doubts about the effects of a cut. We will try to move a single cut around in our example of section 1.3.2:

```
p :- q( X ), write( trying( X ) ), nl,
    female( X ), write( ok ), nl.
p :- write( 'Sorry!' ), nl.

q( X ) :- writer( X ).
q( ? ) :- write( 'No more writers.' ), nl.

writer( hesse ).
writer( mann ).
writer( grass ).
```

```
female( austen ).
female( sand ).
```

If we insert a cut into the first clause of `writer`:

```
writer( hesse ) :- !.
```

the printout will be

```
trying( hesse )
No more writers.
trying( ? )
Sorry!
```

If we insert it into `q` instead:

```
q( X ) :- !, writer( X ).
```

we will get

```
trying( hesse )
trying( mann )
trying( grass )
Sorry!
```

But if we choose to insert it at the end of this clause:

```
q( X ) :- writer( X ), !.
```

the program will write

```
trying( hesse )
Sorry!
```

By inserting the cut after the call to `q` in the first clause of `p`, we would obtain only

```
trying( hesse )
```

As evidenced by these examples, the cut is a powerful tool. A single cut can drastically alter the behaviour of a program. It must be used very carefully: Section 4.3.1 contains some useful hints.

An important property of the cut is that it can be used to implement a sort of negation. When we want to list all male writers, we can write

```
:- writer( X ), male( X ), write( X ), nl, fail.
```

If, however, the program contains only descriptions of female persons (as in our example), we must define *male* in terms of *female*:

```
male( X ) :-female( X ), !, fail.
male( _ ).
```



When *X* is such that *female* succeeds, the second clause of *male* is cut off and the whole procedure fails. When *female* fails, the second clause takes over and the procedure succeeds. The trick is dirty, but very useful. One must be careful, however: if the constant *christie* is not listed among the females, *male(christie)* will succeed. (More on this in Section 4.3.2.)

### 1.3.7. Variable Calls

The negation schema shown in the previous section is of quite general utility. For example, we could write a procedure for checking that two sets (represented as lists) do not intersect:

```
disjoint( S1, S2 ) :- intersect( S1, S2 ), !, fail.
disjoint( _, _ ).
```

Prolog provides a very convenient extension which allows us to use such schemas without going to the trouble of rewriting them again and again. A **variable call** is a variable occupying the position of a call in a clause or directive. When the turn comes to execute the call occupying this position, the variable's current instantiation is taken as the call, by treating its main functor as a predicate symbol and its arguments as parameters. If we define

```
do( X ) :- X
```

then the call

```
carcdr( el.[], A, B )
```

is exactly equivalent to

```
do( carcdr( el.[], A, B ) )
```

as well as to

```
do( do( carcdr( el.[], A, B ) ) ) .
```

We can use this feature to define

```
not( X ) :- X, !, fail.
not( _ ).
```

and write

```
male( X ) :- not( female( X ) ).
disjoint( S1, S2 ) :- not( intersect( S1, S2 ) ).
```

The dirty trick is now nicely packaged.

In versions of Prolog described here, **not** is predefined and the predicate symbol is predeclared as a prefix symbol. Expanding **male** in-line, we would write the directive of Section 1.3.6 as

```
:- writer( X ), not female( X ), write( X ), nl, fail.
```

Variable calls can be used to define many useful procedures. We shall end by showing two companions of “not”: “and” and “or.” The first is written as a comma and the second as a semicolon; the first succeeds when both its parameters—taken as calls—succeed, and the second succeeds when either of its parameters succeeds (but establishes a fail point if it is the first one). Their definitions are

```
','( A, B ) :- A, B.
```

```
','( A, _ ) :- A.
```

```
','( _, B ) :- B.
```

Comma and semicolon are predeclared as infix symbols. After defining *do*, we could write the directive above as

```
:- do( ( writer( X ), not female( X ), write( X ), nl, fail ) ).
```

The extra parentheses are needed to avoid confusion with a call to *do/5*. Priorities are chosen so that

```
artwork( X, Y ) :- painting( X, Y ), oil( Y );
                  etching( X, Y ), brass( Y ).
```

is equivalent to

```
artwork( X, Y ) :- ','( ','( painting( X,Y ),oil( Y ) ),
                        ','( etching( X,Y ),brass( Y ) ) ).
```

To make the comma and semicolon appear a part of Prolog’s syntax, Prolog-10 and some of its offsprings made the cut behave somewhat differently for these procedures: they are “transparent” to it. Thus

```
artwork( X, Y ) :- painting( X, Y ), oil( Y ), ! ;
                  etching( X, Y ), brass( Y ).
```

avoids checking the second alternative if the first succeeds.

A similar exception applies to variable calls. If the procedure

```
a( X ) :- b, X.
a( _ ) :- c.
```



is called with

`a( ( d, !, fail ) )`

then the cut will commit all choices made by `d` and `b` and `a`—the procedure will fail without executing `c`.

One should avoid taking advantage of this peculiar property of the cut. It is doubtful whether it is necessary.

---

## 2 PROLOG AND LOGIC

---

### 2.1. INTRODUCTION

“Prolog” stands for “**P**rogrammation en **l**ogique” (programming in logic). Static interpretation of procedures (see Section 1.3.4) is possible because Prolog can also be viewed as a system for proving theorems expressed in logic. Adopting this viewpoint can provide the programmer with new insights about the nature of his task.

In this chapter we attempt to introduce the fundamentals of this aspect of Prolog in an intuitive manner. Full appreciation of the subject is possible only for people with a solid background in mathematical logic, and we assume your knowledge of logic is very elementary. Consequently, the presentation is often not sufficiently precise, and sometimes the terminology is a little unconventional: we are interested in Prolog rather than logic. The chapter is a shortcut, so in some places you will find it heavy going. A more detailed, but still non-technical treatment can be found in Kowalski (1979b). Another relevant book is Robinson (1979). See also van Emden and Kowalski (1979).

### 2.2. FORMULAE AND THEIR INTERPRETATIONS

Below is a pair of formulae written in the language of predicate logic (also known as first-order logic or predicate calculus):

$$(2.1) \quad \forall x D(Z, x, x)$$

$$(2.2) \quad \forall x \forall y \forall z D(x, y, z) \Rightarrow D(S(x), y, S(z)).$$



The basic building blocks of such formulae are **predicates**. A predicate consists of a **predicate symbol** (e.g.  $D$ ), optionally followed by arguments—a list of **terms** in parentheses, separated by commas. A term is a variable (e.g.  $x, y, z$ ), or a functor (e.g.  $Z, S$ ) with an optional list of arguments, which are terms. Terms denote objects in some universe (more on this presently) and predicates stand for relations between these objects.

A single predicate is a formula. A larger formula can be built from simpler ones by means of **logical connectives**. The commonly used connectives, listed in order of decreasing priority, are

- the **negation** (“not”), written as  $\neg$
- the **conjunction** (“and”), written as  $\wedge$
- the **disjunction** (“or”), written as  $\vee$
- the **implication**, written as  $\Rightarrow$

Parentheses can be used to increase clarity or override priority.

A formula (i.e. also a subformula) can be prefixed by a number of quantifiers, whose priority is lower than that of the connectives. A quantifier can be

- the **existential quantifier**, written as  $\exists x$  and read as “there exists an  $x$ ”.
- the **universal quantifier**, written as  $\forall x$  and read as “for all  $x$ ”, or “for any  $x$ ”.

The formula prefixed by a quantifier is called its **scope**, and the quantified variable is local to this scope (an occurrence of its name outside the scope does not denote the same object). In this chapter we shall deal only with **fully quantified** formulae; i.e. our formulae will not contain unquantified variables.

Our example formulae can be read as

“for any object—call it  $x$ —the object  $Z$  is in relation  $D$  with  $x$  and  $x$ ”

and

“for any three (not necessarily distinct) objects—call them  $x, y$  and  $z$ —if  $x, y$  and  $z$  are in relation  $D$ , then so are objects  $S(x)$ ,  $y$  and  $S(z)$ ”.

In practice, it is more convenient to use a slightly abbreviated reading, in which the second formula is

“for all  $x, y$  and  $z$ ,  $D(x, y, z)$  implies  $D(S(x), y, S(z))$ ”.

Formulae of this kind are purely formal statements. One cannot discuss whether they are true or false, because no particular meaning is attributed to the functors and predicate symbols. To talk about a formula's meaning, we must give it an **interpretation**. An interpretation is a



definition of a **universe** (the set of objects which can be denoted by terms) and a decision to let predicate symbols and functors denote particular relations and functions defined in this universe.

A concrete interpretation maps a (fully quantified) formula to a statement which is true or false, depending on what it says about relations between objects. Somewhat imprecisely, we shall say that a formula is **true (or false) in an interpretation**. Of course, some formulae are true in all interpretations (the formula *true* is a trivial example, and  $A \vee \neg A$  is another); others are false in all interpretations (e.g. *false*,  $A \wedge \neg A$ ). The first kind of formulae are called tautologies; formulae of the second kind are called **inconsistent**.

As an example, consider the following two interpretations of formulae (2.1) and (2.2). The first interpretation is the following:

- the universe is the set of natural numbers (positive integers);
- $Z$  stands for the number 1 (one);
- $S$  stands for the function  $S(x) = 2x$ ;
- $D(x, y, z)$  is true if and only if  $xy = z$ .

Our formulae now become the true statements

“for any natural number  $x$ ,  $1x = x$ ”

and

“for all natural numbers  $x, y$  and  $z$ ,  $xy = z$  implies  $2xy = 2z$ ”.

Another interpretation is:

- the universe is the set of non-negative integers;
- $Z$  stands for the number 0 (zero);
- $S$  stands for the successor function  $S(x) = x+1$ ;
- $D(x, y, z)$  is true if and only if  $x+y = z$ .

The formulae are now

“for any non-negative integer  $x$ ,  $0+x = x$ ”

and

“for all integers  $x, y$  and  $z$ ,  $x+y = z$  implies  $(x+1)+y = z+1$ ”.

If an interpretation maps a formula into a true statement, then this interpretation is called a **model** of this formula. We can also speak about a model of a **set** of formulae—an interpretation in which all of them are true.

Our two interpretations are models of the example formulae. If  $Z$  stood for 1 in the second interpretation, then it would not be a model. When an interpretation interests us as a model, formulae which are true (or false) in that interpretation will be referred to as true (or false) **in the model**.



All interpretations are models of tautologies. Inconsistent formulae have no models.

When we want to talk about a particular model, we prefer to use symbols which have some mnemonic value. The formula

$$(2.3) \quad \forall h \forall t \text{ conscarcdr}(. (h, t), h, t)$$

can be interpreted as

“for all integers  $h$  and  $t$ , the difference between  $h+t$  and  $h$  is  $t$ ”,

but this is better written as

$$\forall h \forall t \text{ difference}(+(h, t), h, t).$$

The similarity is interesting, though: looking for other models of our statement of a problem often provides illuminating insights into its nature.

Notice that the “natural” interpretation of formula (2.3) is very down-to-earth. A list constructor can be thought of as a function mapping two objects (a head and a tail) into a third object: the universe can be a set of data structures.

## 2.3. FORMAL REASONING

The notion of **logical consequence** allows us to perform **formal** reasoning, i.e. reasoning which takes into account only the syntactic form of formulae and disregards their interpretations. We say that formula  $\alpha$  is a logical consequence of a set of formulae  $\beta, \beta', \beta'' \dots$  if all models of the set  $\beta, \beta', \beta'' \dots$  are also models of  $\alpha$ . It is a fundamental fact of logic that there exist **inference rules**, which are correct recipes for deriving logical consequences (conclusions) of other formulae (premises), provided the latter have a certain form. The inference rules are usually quite simple, but we can use them as elementary steps in long derivations. This is the backbone of mathematics: a set of formulae (**axioms**) defines a **theory**, which is the set of all formulae (called **theorems**) true in all models of the axioms; a formal derivation of a new theorem is called its **proof**. (The name **axioms** is often reserved for a **minimal** set of theorems specifying the theory of interest. We find it more convenient to use the name for any “given” set of theorems accepted without proofs.)

Some inference rules are relatively trivial applications of the definitions of logical connectives. A well-known example is the **modus ponens**:

“from any formula  $\alpha$  and from any formula of the form  $\alpha \Rightarrow \beta$ , derive the formula  $\beta$ ”.



Now the definition of implication can be stated as follows: if  $\alpha$  and  $\beta$  are arbitrary formulae, then, *in any interpretation*,  $\alpha \Rightarrow \beta$  is false if and only if  $\alpha$  is true and  $\beta$  is false in that interpretation. Hence, any model of both  $\alpha$  and  $\alpha \Rightarrow \beta$  must also be a model of  $\beta$ .

Two other simple rules are

“ $\alpha \Rightarrow \beta$  is equivalent to  $\neg \alpha \vee \beta$ ,  
(i.e. one can be derived from the other)”

and one of the De Morgan laws

“ $\neg (\alpha \wedge \beta)$  is equivalent to  $\neg \alpha \vee \neg \beta$ ”.

Do convince yourself of their validity—we will need them presently!

Armed with a number of inference rules, we can attempt to derive a formula directly or by means of a technique known as **reductio ad absurdum**. To derive formula  $\alpha$  from a set of axioms, assume that  $\neg \alpha$  is a theorem: if the resulting theory is inconsistent, then  $\alpha$  is a theorem. A theory is inconsistent if it contains an inconsistent formula. In this method of proof we often show inconsistency by finding a formula  $\beta$  such that we can derive

$$\beta \wedge \neg \beta.$$

It is worth noting that all formulae are theorems of an inconsistent theory. This is because, there being no models of the theory, no formula is false in any of the models. (This might not have sounded too convincing, but notice that if we can derive *false*, then we can derive any formula  $\alpha$  using **modus ponens** and  $\text{false} \Rightarrow \alpha$ . For any  $\alpha$ , the formula  $\text{false} \Rightarrow \alpha$  is a tautology, because it is equivalent to  $\neg \text{false} \vee \alpha$ , that is to say  $\text{true} \vee \alpha$ .) Consequently, if the set of formulae

$\neg \alpha$

$\beta$

$\beta'$

...

is inconsistent, then  $\alpha$  is certainly a theorem, regardless of whether the set  $\beta, \beta', \dots$  is consistent or not.

## 2.4. RESOLUTION AND HORN CLAUSES

We shall be interested in an inference rule which we shall call the **rule of resolution** (Robinson 1965). It says

“from  $\neg \alpha \vee \beta$  and from  $\alpha \vee \gamma$  derive  $\beta \vee \gamma$ ”.



Its validity is not hard to explain. In any model of  $\neg \alpha \vee \beta$  and  $\alpha \vee \gamma$ , either  $\neg \alpha$  is false or  $\alpha$  is false. In the first case  $\beta$  must be true (or else  $\neg \alpha \vee \beta$  would not be true), in the second  $\gamma$  must be true. If a model of  $\neg \alpha \vee \beta$  and  $\alpha \vee \gamma$  must also be a model of  $\beta$  or a model of  $\gamma$ , then—by definition of disjunction—it is a model of  $\beta \vee \gamma$ .

There are two interesting special cases of this rule. One is

“from  $\neg \alpha \vee \beta$  and from  $\alpha$  derive  $\beta$ ”,

and the other is

“from  $\neg \alpha$  and from  $\alpha$  derive  $\square$ ”.

Here  $\square$  stands for the empty formula, which must be treated as equivalent to *false* if this form of the rule is to be valid.

The rule of resolution is useful for reductio ad absurdum proofs when our formulae are written in a restricted form called clausal form. A **clause** is a disjunction of **literals**. A literal is either a predicate (called **positive literal**) or a negated predicate (called **negative literal**). All clauses are prefixed by universal quantifiers, one for each variable in the clause.

We shall limit our attention to **Horn clauses**, which have at most one positive literal each. Here is a set of four Horn clauses (the predicates are all nullary):

$A \vee \neg B \vee \neg C$

$B \vee \neg D$

$C$

$D$

Now, if we want to prove that  $A$  can be derived from these clauses, we can use the rule of resolution to show that by adding the Horn clause

$\neg A$

to our set of formulae, we obtain an inconsistent set of clauses. The proof can be carried out in the following four steps (we use parentheses to make things more clear):

1. from  $\neg A$  and from  $A \vee (\neg B \vee \neg C)$  derive  $\neg B \vee \neg C$
2. from  $\neg B \vee \neg C$  and from  $B \vee \neg D$  derive  $\neg C \vee \neg D$
3. from  $\neg C \vee \neg D$  and from  $C$  derive  $\neg D$
4. from  $\neg D$  and from  $D$  derive  $\square$

Notice that this type of reductio ad absurdum proof is successful when we derive the empty clause  $\square$  (i.e. *false*). The special cases of the resolution rule are used to shorten formulae, while the general rule is used



to generate formulae which can be shortened. Now, if in an application of the resolution rule both the premises have one positive literal each, then the conclusion must also have one positive literal (do you see why?). Hence, the proof cannot be successful unless at least one of the clauses has no positive literals. However, if one of the premises has only negative literals, then so has the conclusion. If only one of the initial clauses has this form, then the proof can be made particularly simple (Kowalski and Kuehner 1971; Hill 1974). One of the premises in the first step is the clause without positive literals. If this step cannot derive the empty clause, then the second step must use the only other clause without positive literals, i.e. the conclusion of the preceding step, and so on. If—as in the example—all our axioms have positive literals, then the negated theorem must have none and the final proof has the form of an orderly chain, in which each step provides a premise for the immediately succeeding one. In each step, we shall call the premise without positive literals the **current resolvent**.

Each step consists in **cancelling** a negative literal  $\neg\lambda$  in the current resolvent, by replacing it with the negative literals of a clause having  $\lambda$  as its only positive literal. The resolvent shrinks when one of its literals is cancelled with a **unit clause**, which has only a single positive literal.

Because a clause is a disjunction of literals, it can be written as an implication. By the De Morgan law (see Section 2.3)  $A \vee (\neg B \vee \neg C)$  is equivalent to  $A \vee \neg (B \wedge C)$ . This, in turn, is equivalent to  $B \wedge C \Rightarrow A$ .

By analogy, we can write

$$B \wedge C \Rightarrow$$

to denote  $\neg B \vee \neg C$  (i.e. a Horn clause with no positive literals). The empty consequent represents *false*, since *false* (or its equivalent) is the only formula  $\alpha$  such that  $\alpha \vee (\neg B \vee \neg C)$  is equivalent to  $\neg B \vee \neg C$ , for all  $B$  and  $C$ .

Similarly, we shall denote  $A$ , a Horn clause with no negative literals, by

$$\Rightarrow A$$

Here, the empty premise represents *true*:  $A$  is equivalent to  $true \Rightarrow A$ . An empty clause has no literals and can be written

$$\Rightarrow$$

i.e.  $true \Rightarrow false$ , which is equivalent to *false*.

It is preferable to write the implication from right to left:

$$A \Leftarrow B \wedge C$$



to suggest the reading

“to prove A, prove B and C”.

Recalling that our resolvents have no positive literals, we can now write the resolution rule as

“from  $\Leftarrow \alpha \wedge \beta$  and from  $\alpha \Leftarrow \gamma$  derive  $\Leftarrow \gamma \wedge \beta$ ”

where  $\beta$ ,  $\gamma$  or both may be empty (in that case we do not write a  $\wedge$ ). This is a rather mechanical prescription: to get rid of  $\alpha$ , find an implication whose consequent is  $\alpha$  and replace  $\alpha$  with its premises. This is clearly justified: since  $\alpha$  can be proven by proving  $\gamma$ , then  $\alpha \wedge \beta$  can be proven by proving  $\gamma \wedge \beta$ .

A clause being always prefixed with universal quantifiers for each of its variables, it is convenient not to write the quantifiers. Our formulae from Section 2.2 are two Horn clauses, written as

$$\begin{aligned} D(Z, x, x) &\Leftarrow \\ D(S(x), y, S(z)) &\Leftarrow D(x, y, z) \end{aligned}$$

Let us see whether these clauses are consistent with

$$\Leftarrow D(S(Z), x, S(S(Z)))$$

The clause can be thought of as a query whether there exists an  $x$  which is in relation  $D$  with  $S(Z)$  and  $S(S(Z))$ . It is equivalent to

$$\forall x \neg D(S(Z), x, S(S(Z))),$$

and since this is the *negation* of what we are trying to prove, our derived formula—if we succeed—will be

$$\exists x D(S(Z), x, S(S(Z)))$$

(if  $\alpha$  is not false for all  $x$ , then there must be at least one  $x$  for which  $\alpha$  is true).

The rule of resolution, in the form presented above, is useless for this example. In fact, it could only be used for nullary predicates, because the argument for its validity does not apply to premises such as

$$\forall x \neg A(x) \vee \neg B(x) \quad \text{and} \quad \forall y A(Z) \vee \neg C(y).$$

Fortunately, we can also employ a simple inference rule called the **substitution rule**. It says

“from  $\forall x \alpha(x)$  derive  $\alpha(\tau)$ , where  $\tau$  is an arbitrary term”.

Here,  $\alpha(x)$  means that the formula  $\alpha$  contains occurrences of variable  $x$ .  $\alpha(\tau)$  stands for a formula which looks exactly like  $\alpha$ , except that all occur-



rences of  $x$  have been replaced by occurrences of term  $\tau$ . An example of the rule's application is

“from  $\forall x D(S(Z), x, S(x))$  derive  
 $D(S(Z), S(S(Z)), S(S(S(Z))))$ ”

The substitution rule is valid, of course. In every model of  $\forall x \alpha(x)$ ,  $\alpha$  is true for any object  $x$  (that is what the quantifier says!): hence, it is true for any particular object.

It should now be clear that

“from  $\forall x \alpha(x)$  and  $\forall y \beta(y)$  derive  $\forall z \alpha(z) \wedge \beta(z)$ ”

is also a valid inference rule. It can be looked on as an application of the substitution rule: in all models of  $\forall x \alpha(x)$  and  $\forall y \beta(y)$ ,  $\alpha(\tau)$  and  $\beta(\tau)$  are true for any  $\tau$ , hence (by the definition of conjunction)  $\alpha(\tau) \wedge \beta(\tau)$  is true for any  $\tau$ , and therefore  $\forall z \alpha(z) \wedge \beta(z)$  is true.

The substitution rule allows us to **match** different formulae by using appropriate variable substitutions. For example, we can easily show that  $\neg D(x, y, z)$  is inconsistent with  $D(Z, Z, Z)$ , because we can derive  $\neg D(Z, Z, Z)$  from the first formula by substituting  $Z$  for  $x, y$  and  $z$ . We shall denote such substitutions by

$x \leftarrow Z, y \leftarrow Z, z \leftarrow Z$ .

Our ability to match formulae allows us to apply to the problem at hand implications which express general rules. This is best illustrated with our running example. (We use apostrophes to distinguish between variables similarly named, but quantified in different scopes or used in different applications of a formula.)

1. Match  $\Leftarrow D(S(Z), x, S(S(Z)))$   
 and  $D(S(x'), y', S(z')) \Leftarrow D(x', y', z')$   
 by substituting  $x' \leftarrow Z, y' \leftarrow x, z' \leftarrow S(Z)$ .
2. From  $\Leftarrow D(S(Z), x, S(S(Z)))$   
 and  $D(S(Z), x, S(S(Z))) \Leftarrow D(Z, x, S(Z))$   
 derive  $\Leftarrow D(Z, x, S(Z))$  by the rule of resolution.
3. Match  $\Leftarrow D(Z, x, S(Z))$  and  $D(Z, x'', x'') \Leftarrow$   
 by substituting  $x'' \leftarrow x, x \leftarrow S(Z)$ .
4. From  $\Leftarrow D(Z, S(Z), S(Z))$  and  $D(Z, S(Z), S(Z)) \Leftarrow$   
 derive  $\square$  (the empty resolvent) by the rule of resolution.

A noteworthy feature of this example is that the term  $S(Z)$ , finally substituted for the  $x$  in

$\Leftarrow D(S(Z), x, S(S(Z)))$



can be thought of as a **counterexample** to the disproved hypothesis that this formula is consistent with the others. We learned in effect that, in any model of the two clauses playing the role of axioms,

“it is not true that  $\forall x \neg D(S(Z), x, S(S(Z)))$ , because  $\neg D(S(Z), x, S(S(Z)))$  is false when  $x$  is  $S(Z)$ ”.

But this is the same as saying that our question

“does there exist an  $x$  such that  $D(S(Z), x, S(S(Z)))$ ”

is answered with

“yes,  $S(Z)$  is such an  $x$ ”.

Recall our two interpretations from Section 2.2. In the first we asked whether there is an  $x$  such that  $2x = 4$ , and the answer is that 2 is such an  $x$ . In the second interpretation, the question whether there is an  $x$  such that  $1 + x = 2$  was answered by 1.

The various substitutions were used to narrow the set of interesting objects to those objects for which the formula being disproved is not true. Indeed, it is evident that for all  $x$  other than  $S(Z)$ , the formula  $\neg D(S(Z), x, S(S(Z)))$  is true in both interpretations. It is so in all models of the two original formulae, but we shall not attempt to justify it directly. Our example would be an “indirect” justification if we could be certain that the substitutions did not “lose” other objects satisfying the disproved formula. Such certainty would be of practical value, because the answer to our query (i.e. our counterexample) can be a term containing variables. We want it to be as general as possible, in the sense that the set of all terms obtainable by substituting something for its variables should be the set of all answers.

It is a fundamental fact of resolution theory that the algorithm of unification (as presented in Section 1.2.3, but extended with the occur check) finds the most general set of substitutions needed to match two literals. “Most general” means that it is contained by all sets of substitutions which make the literals match. When we match  $A(x)$  and  $\neg A(y)$ , both  $x \leftarrow y$  and  $y \leftarrow x$  are possible—we treat them as indistinguishable. In a sense, this is the minimal necessary set of substitutions.

We used unification in our example, so we did not lose any solutions. Notice, however, that our discussion is concerned with the effects of the substitution rule; as we shall see, different proofs can come up with different solutions. Try the *conscardr* and *append* examples from Chapter 1 to get a feeling for this kind of proof. You may use infix notation for functors—it does not matter. The *intersection* example might be a little more difficult: read on.



## 2.5. STRATEGY

Disjunction being commutative, we can apply the rule of resolution to any literal in the current resolvent: in our examples, we always chose the leftmost one. The choice does not affect our ability to finish the proof, as we must be able to cancel all the literals before obtaining the empty resolvent. As it turns out, the desirable properties of unification mentioned in the previous section ensure that the order in which the cancelling of literals is performed does not influence the final outcome of the proof. The *length* of a proof, however, can be affected by the choice of literals very strongly indeed. We shall discuss this matter at the end of this section.

In our examples, at most one clause could be used to cancel a literal in each step. In general, a number of clauses can be applicable (after suitable matching) to a given literal. Choosing the right clause could be important, because some of them can lead into "blind alleys". After many steps, we may turn up with a resolvent to which the rule of resolution cannot be applied (because there are no matching clauses for its literals), even though another choice of clause at an earlier step might have speedily led to the empty resolvent.

The situation is illustrated by Fig. 2.1b, which shows part of the **search space** for the problem listed in Fig. 2.1a. The space is tree-shaped: each path from the root to a leaf represents a possible derivation sequence; its nodes are labelled with the successive resolvents. Some of the paths are successful, some end in failure.

Notice that several subtrees occur more than once. This effect would be less pronounced if the resolvents reflected the history of substitutions, but we did not feel up to creating such a drawing for predicates with arguments. Try it for the application of *intersect* traced in Section 1.3.3. (Figure 2.5 will show you how to do it.)

Prolog always tries to use the leftmost literal, so its search space is considerably smaller, as illustrated in Fig. 2.1c. Whenever it is presented with a number of applicable clauses, the system always attempts the first one first. When it encounters failure, it backtracks and tries another path. In effect, it executes an orderly preorder search of the search space tree.

Figure 2.1c also illustrates the effect of a cut: a part of the search space is shorn off, but one must be aware that this part may contain solutions! While this is not important in the example (if we expect a yes/no answer), in general different solutions may represent essentially different instantiations (i.e. substitutions) of variables in the root of the tree.

This is not in contradiction to our earlier statements about "the desirable properties of unification". As evidenced by Fig. 2.2, by choosing



different literals we change only the order in which things are proved, but the general structure of the proof is not changed. It is convenient to represent the structure of a proof by means of a **proof tree** (do not confuse it with the search space), as illustrated in Fig. 2.3. The first tree shows the proofs of the preceding figure, which all used B and C to prove A, and F to prove B. The other proof trees represent classes of proofs obtained through a different choice of clauses: the proofs are carried out quite differently.

Structurally different proofs use different subsets of the available clauses for performing various subproofs, so the “counterexamples” of Section 2.4—which are descriptions of sets of objects for which the clauses used cannot all be true—may turn out to be different. Therefore, not all solutions are the same.

Proof trees are interesting also because they reflect the invocation tree when clauses are treated as procedures. As long as there are no failures, the conventional procedure activation stack can be regarded as an equally conventional stack used for preorder traversal of the proof tree, or—if failure is imminent—of a quasi-proof tree which has a failure

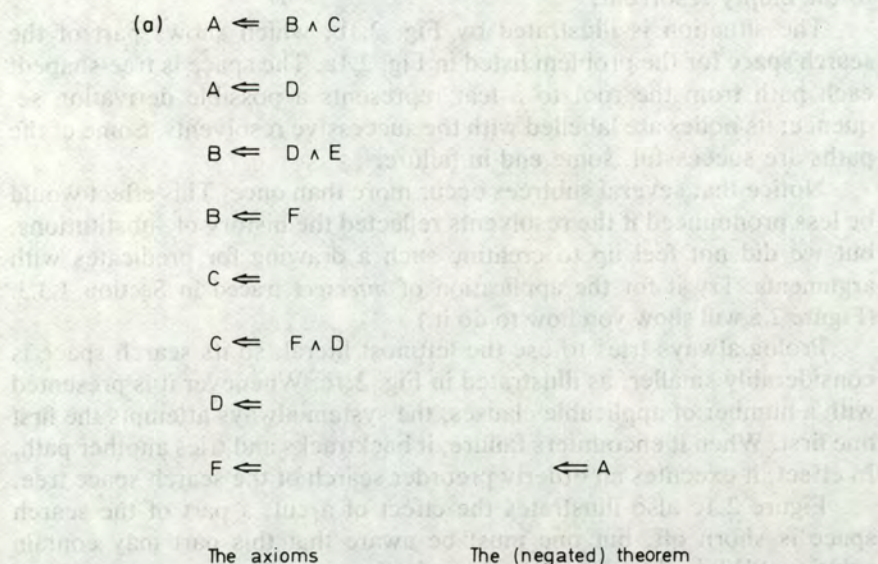


FIG. 2.1 (a) A search space: the source formulae. (b) A search space: an initial part of the complete space. (Choice of literals denoted by an arc, choice of clauses by a dot.) (c) A search space: Prolog search space has no choices for literals. (Solution one is found, the others could be found on backtracking. ! marks the subspaces made unreachable by executing a cut at the end of A's first clause.) (continued)

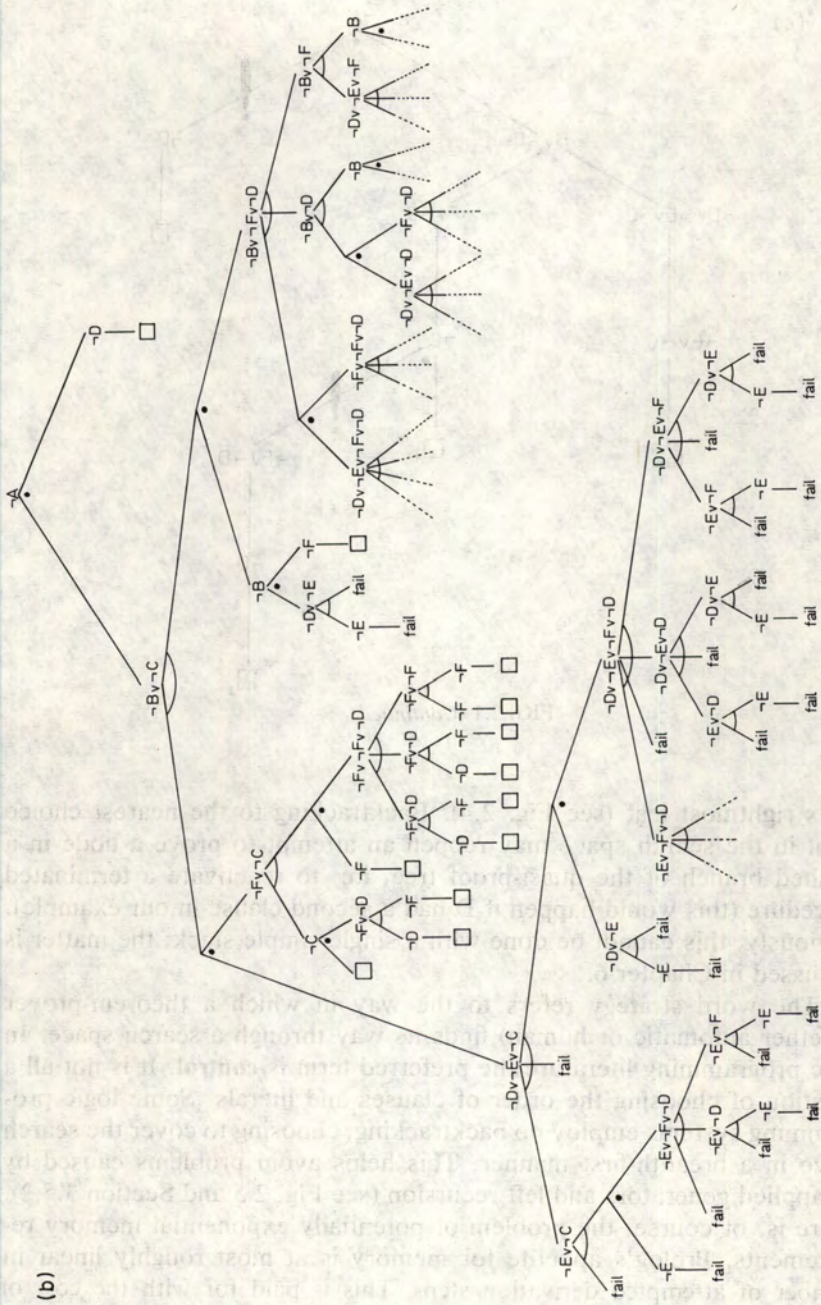


FIG. 2.1 (Continued)



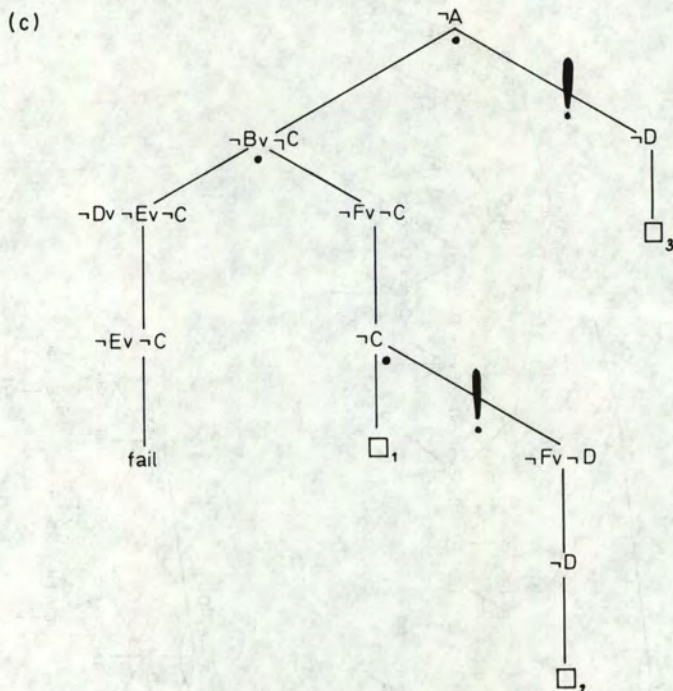
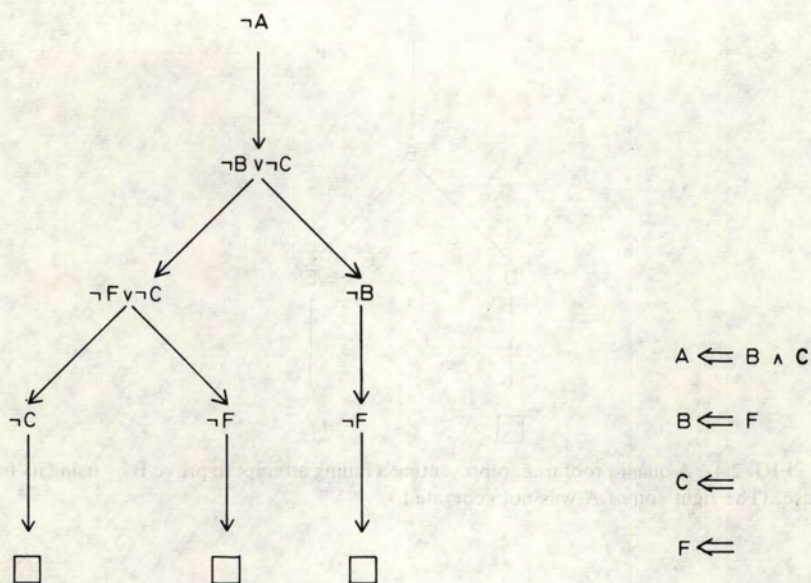


FIG. 2.1 (Continued)

in its rightmost leaf (see Fig. 2.4). Backtracking to the nearest choice point in the search space may reopen an attempt to prove a node in a finished branch of the quasi-proof tree, i.e. to reactivate a terminated procedure (this would happen if  $D$  had a second clause in our example). Obviously, this cannot be done with a single simple stack: the matter is discussed in Chapter 6.

The word **strategy** refers to the way in which a theorem-prover (whether automatic or human) finds its way through a search space. In logic programming literature the preferred term is **control**. It is not all a question of choosing the order of clauses and literals. Some logic programming systems employ no backtracking, choosing to cover the search space in a breadth-first manner. This helps avoid problems caused by misapplied generators and left recursion (see Fig. 2.5 and Section 3.5.2). There is, of course, the problem of potentially exponential memory requirements. Prolog's appetite for memory is at most roughly linear in number of attempted derivation steps. This is paid for with the cost of backtracking: the time complexity is still exponential.



The tree of possible proof paths

The clauses used in these proofs

FIG. 2.2 Choice of literals with fixed choice of clauses from Fig. 2.1. (Prolog would choose the leftmost path, but only after some backtracking caused by choosing the first clause of B.)

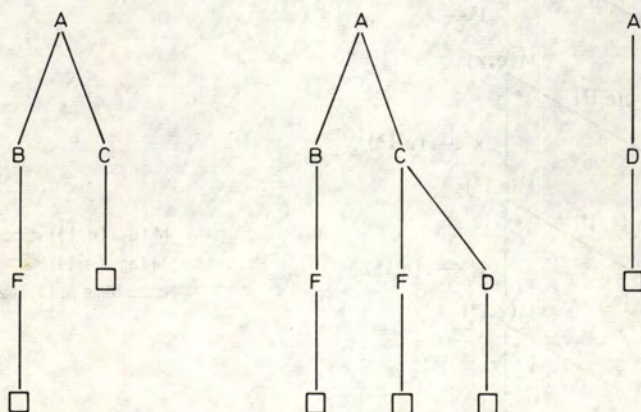


FIG. 2.3 Proof trees for the example of Fig. 2.1. (The leftmost tree represents the proofs of Fig. 2.2. Backtracking would cause Prolog to build each tree in turn, from left to right.)



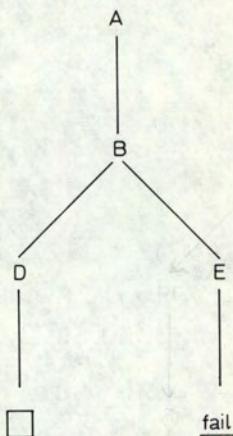


FIG. 2.4 A quasi-proof tree, representing a failing attempt to prove B by using its first clause. (The right son of A was not generated.)

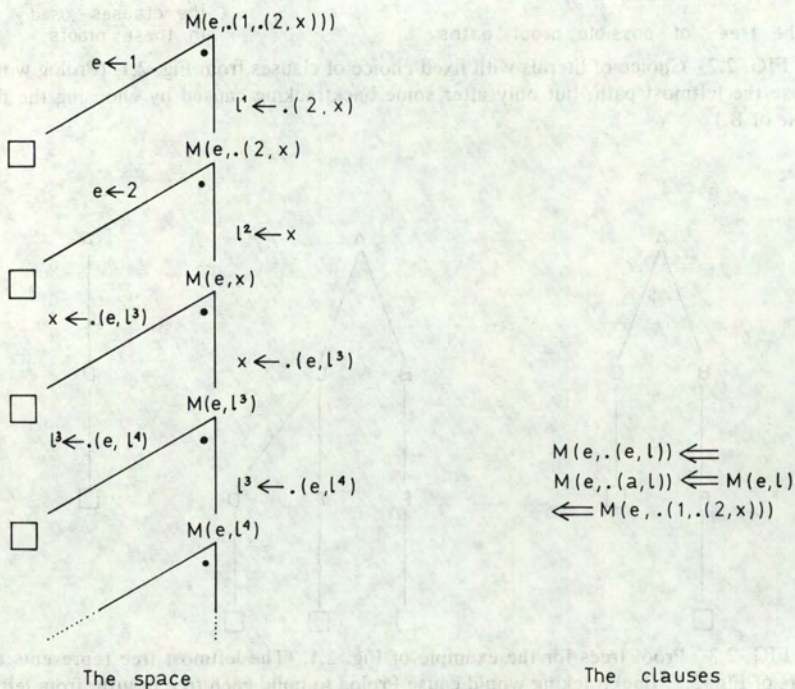


FIG. 2.5 Prolog search space for an infinite generator. (The example is *member*.)

There have been attempts to decrease this cost by means of a more sophisticated backtracking strategy (Bruynooghe 1978, Pereira and Porto 1980b, 1982, Bruynooghe and Pereira 1981).

The more ambitious scheme, appropriately called "intelligent backtracking", attempts to retain subproofs (which would otherwise have been discarded) in order to avoid recomputing them again and again. In other words, it attempts to take advantage of the multiplicity of identical subtrees in a search space (compare Fig. 2.1b).

A simpler approach, called "selective backtracking", consists in analysing which variable instantiations caused the failure. It is then possible to backtrack directly to the nearest point where one of these instantiations was made or where the computation would take an entirely different course. In some cases this can save us a lot of thrashing about in a failure-infested region of the tree's crown.

Unfortunately, these interesting ideas have not influenced Prolog implementations. They require a further complication of the already complex runtime data structures and they do not mesh well with side-effects of system procedures. The fact that Prolog can be used as a practical language is still largely due to our dexterity in fighting exponential complexity with the cut.

Attempts to modify Prolog's strategy so that it would incorporate parallelism or coroutining have been a little more successful. Parallelism consists in growing various branches of a proof tree (or even several trees) simultaneously. It is difficult, not only because it raises tricky technical problems, but because we still lack sufficient understanding of its effects on both time/space complexity and the number of solutions gained or lost. Several very different approaches have been documented, but none of them seems to answer all pertinent questions. Some of the references are (Clark and Gregory 1983, Shapiro 1983b, Conery and Kibler 1983, Wise 1984, Eisinger *et al.* 1982).

Coroutining is just that: switching control between several active procedures. In terms of our drawings, coroutining is a non-trivial traversal of a proof tree. Roughly, it is a matter of choosing a different order of literals (a different path in Fig. 2.2). It is possible to demonstrate spectacular improvements in the performance of some programs when they are executed in coroutining fashion. A simple example is the naive naive sort (see Section 1.3.5):

```
sort( List, Sorted ) :- permute( List, Sorted ),
                        ordered( Sorted ).
```

When the execution of *permute* is interleaved with that of *ordered* so that the latter can cause failure as soon as the first out-of-order element is



produced by the former, the program's behaviour compares very favourably with that shown when each permutation must be completed before *ordered* is called. When the initial sequence of a permutation is rejected, all other permutations starting with the same sequence can at once be rejected as well, resulting in a significant reduction of the search space.

Section 9.2 contains two short examples of coroutining Prolog programs. Coroutining comes in many flavours: some of the references are (Clark *et al.* 1979, Clark *et al.* 1982, Porto 1982, Colmerauer *et al.* 1983). Unfortunately, most of these schemes are of restricted utility (Kluźniak 1981). The problem is that coroutines do not mesh well with backtracking. We comment on this at greater length in (Kluźniak and Szpakowicz 1984).

---

## 3 METAMORPHOSIS GRAMMARS: A POWERFUL EXTENSION

---

### 3.1. PROLOG REPRESENTATION OF THE PARSING PROBLEM

We shall begin with a very simple formulation of the parsing problem: given a sequence of items, find out whether it has some presupposed structure. The problem appears e.g. in programming languages when we want to make sure that some text is a syntactically valid statement. Admissible structures are usually described by a context-free grammar. As an example we shall consider the following small grammar in Backus-Naur-Form, which describes simple list expressions:

(3.1) 
$$\begin{aligned} \langle \text{list} \rangle &::= () \mid ( \langle \text{items} \rangle ) \\ \langle \text{items} \rangle &::= \langle \text{item} \rangle \mid \langle \text{item} \rangle , \langle \text{items} \rangle \\ \langle \text{item} \rangle &::= \langle \text{atom} \rangle \mid \langle \text{list} \rangle \\ \langle \text{atom} \rangle &::= \langle \text{letter} \rangle \mid \langle \text{letter} \rangle \langle \text{atom} \rangle \\ \langle \text{letter} \rangle &::= a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid \\ &\quad n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z \end{aligned}$$

Terminal symbols of this grammar are small letters, round brackets, and a comma. For example, the list

(a,(big,ox))

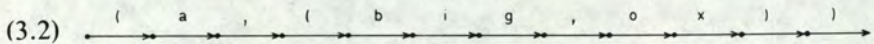
consists of 12 terminal symbols.

There are several commonly used methods of describing the structure of a list (or, more generally, of a valid sequence of terminal symbols). The



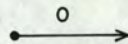
method we adopt here leads to an elegant formulation of the parsing problem in Prolog<sup>1</sup>.

We shall depict a sequence of terminal symbols in a graph (3.2):

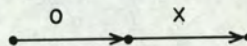


Every node in this graph corresponds to a boundary between two consecutive terminal symbols; every edge connecting two nodes corresponds to the terminal symbol it is labelled with. Two edges are *contiguous* if they share a node; a sequence  $e_1, \dots, e_m$  of edges is *contiguous* if  $e_i$  and  $e_{i+1}$  are contiguous for  $i = 1, 2, \dots, m - 1$ . For example, the edges labelled  $b, i, g$  are contiguous. The labels of contiguous edges are also *contiguous*.

A sequence of contiguous labels may constitute a whole which is meaningful in that it corresponds to the right-hand side of a production. For example, the (only) label of the one-element sequence of edges



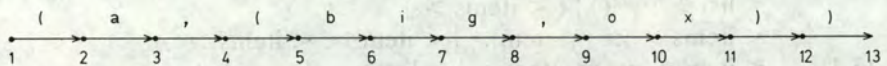
constitutes a letter; the labels of the sequence



constitute an atom.

We shall describe such meaningful combinations by connecting the extreme nodes of a contiguous sequence by an edge. The edge will be labelled with the name of an appropriate non-terminal symbol, as for example in Fig. 3.1.

To be able to represent graphs in a program, we must give each node a unique name. For example, we can name nodes with numbers:



We can represent such a graph as a set of edges, every edge expressed by a unit clause<sup>2</sup> that specifies the label of the edge and the names of the nodes it connects. Perhaps the most compact way is to use the label as the clause name, e.g.

atom( 9, 11 ).

letter( 9, 10 ).

o( 9, 10 ).

<sup>1</sup> This manner of presentation is due to Colmerauer; it was also used by Kowalski (1979b).

<sup>2</sup> For other ways of representing graphs in Prolog, see Sections 4.2.4 and 4.4.3.

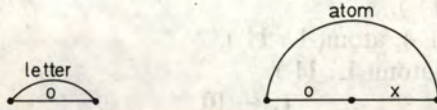


FIG. 3.1 Meaningful combinations of edges.

We now observe that clauses which represent edges labelled with non-terminal symbols might be derived from those corresponding to terminal symbols, by virtue of general structural relationships inherent in the grammar. The reasoning would be roughly as follows:

letter( 9, 10 ) because o( 9, 10 ): o is a letter;  
 letter( 10, 11 ) because x( 10, 11 ): x is a letter;  
 atom( 10, 11 ) because letter( 10, 11 ): a letter makes an atom;  
 atom( 9, 11 ) because letter( 9, 10 ) and atom( 10, 11 ): a letter and  
 an atom make an atom.

Relationships of this kind can be generalized in a straightforward manner, e.g.

(3.3) letter( K, L ) :- o( K, L ).  
 letter( K, L ) :- x( K, L ).  
 atom( K, L ) :- letter( K, L ).  
 atom( K, M ) :- letter( K, L ), atom( L, M ).

Contiguity of edges is assured by using the same term (variable name) to denote every intermediate node: once at the end of an edge and once at the beginning of the next one.

Given the clauses that describe edges with terminal symbols, e.g.

(3.4) o( 9, 10 ).  
 x( 10, 11 ).

we might now derive all the remaining relevant edges. Strictly speaking, they would be present only implicitly. For example, to confirm the presence of the edge

atom( 9, 11 )

we would issue the command

:- atom( 9, 11 ).

from which the following computation might ensue:



```

atom( 9, 11 ).
letter( 9, L ), atom( L, 11 ).
o( 9, L ), atom( L, 11 ).
(3.5)                                     L ← 10
atom( 10, 11 ).
letter( 10, 11 ).
x( 10, 11 ).
success

```

The method of specifying the initial graph is rather awkward, even for this small example. Moreover, it requires that terminal symbols be only identifiers (nullary functors)—the restriction is unnatural but, fortunately, unnecessary. We shall now describe a slightly different and much handier notation.

Names of nodes need not be consecutive integers. On the contrary, it is much better to derive (unique) names from the original sequence of terminal symbols than to introduce another, completely independent nomenclature. We shall exploit the one-one correspondence between a node and the sequence of (contiguous) edges following it. As the name of a node we shall take the *list* of terminal symbols labelling the corresponding sequence. For example, the leftmost node of the graph (3.2) will be named

`'(.a.,'.('b.i.g.','.o.x.')).').[]`

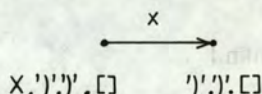
and the name of the rightmost one—corresponding to the empty sequence of nodes—will be

`[]`

With this notation, the (implicit) clause describing the atom ox becomes

`atom( o.x.').').[], ').').[] ).`

Notice how the underlying sequence of terminal symbols can be seen without resorting to separate clauses for o and x: it is simply the “difference” of the first and the second node names, o and x in our case. For a terminal symbol this difference is guaranteed to consist of the symbol itself, as, say, in



In other words, if an edge connects the nodes  $X$ ,  $Y$  and is labelled with the terminal symbol  $T$ , then

$$X = T.Y$$

In order to allow arbitrary terms as terminal symbols, we can write, e.g.

```
terminal( o, K, L )
```

instead of  $o(K, L)$ . Moreover, rather than writing

```
.....
terminal( o, o.x.'').'.[] , x.'').'.[] ).
terminal( x, x.'').'.[] , ''.'').[] ).
```

we shall use the general-purpose, one-clause auxiliary procedure

```
terminal( T, T.Y, Y ).
```

However, now we need some other way of specifying the initial sequence of terminal symbols, which in the previous formulation could be read from the assertions (3.4). Before we explain this, we shall rewrite (3.3):

```
letter( K, L ) :- terminal( o, K, L ).
letter( K, L ) :- terminal( x, K, L ).
atom( K, L ) :- letter( K, L ).
atom( K, M ) :- letter( K, L ), atom( L, M ).
terminal( T, T.Y, Y ).
```

The computation analogous to that shown in (3.5) would now look as follows:

```
atom( o.x.'').'.[] , ''.'').[] ).
letter( o.x.'').'.[] , L , atom( L, ''.'').[] ).
terminal( o, o.x.'').'.[] , L , atom( L, ''.'').[] ).
      L ← x.'').'.[]
(3.6) atom( x.'').'.[] , ''.'').[] ).
      letter( x.'').'.[] , ''.'').[] ).
      terminal( x, x.'').'.[] , ''.'').[] ).
      success
```

All the necessary information about the initial graph was supplied by the first call. What is more, the graph itself is now implicit: we only get—and manipulate—the two sequences of terminal symbols.

We are now in a good position to restate each instance of the parsing problem in terms of Prolog. A grammar is given in the form of Prolog clauses, each clause corresponding to some structural relationship between a unit and its immediate components (in particular, to a BNF rule). For example,

```
items( K, N ) :-
    item( K, L ), terminal( ', ', L, M ), items( M, N ).
```



A call on one of these clauses (or, to be more precise, on the procedure to which it belongs) fully specifies two lists of terminal symbols, the second being the tail of the first. As a matter of convention, the clause name will also be the name of a nonterminal symbol, i.e. it will tell us what structure we want to attribute to the underlying sequence of terminal symbols. For example, the call

```
:- items( b.i.g.', 'o.x.')'.[]', ').''.[] ).
```

can be interpreted as the question: In the graph determined by the parameters, can an edge labelled with *items* be validly drawn between the extreme nodes? Or briefly: Is *items* the valid structure of a given sequence of terminal symbols, big,ox in our case?

The answer to this question is YES if the call succeeds, and NO otherwise. Examples of unsuccessful attempts to parse are;

```
:- items( ', 'o.x.')'.[]', ').''.[] ).  
    /items cannot begin with a comma/  
:- atom( o.x.')'.[]', ').''.[] ).  
    /atom cannot end with a bracket/
```

Procedural interpretation can be expressed in terms of the successive augmentation of the original graph. Every *successful* call implicitly adds an edge. Parsing succeeds if we can connect the extreme nodes with a single edge. This construction proceeds bottom-up: we can imagine an edge being added only after the successful *termination* of a corresponding call.

We shall illustrate this by a complete program for parsing lists.

```
list( K, M ) :- terminal( '(', K, L ), terminal( ')', L, M ).  
list( K, N ) :-  
    terminal( '(', K, L ), items( L, M ), terminal( ')', M, N ).  
items( K, L ) :- item( K, L ).  
items( K, N ) :-  
    item( K, L ), terminal( ',', L, M ), items( M, N ).  
(3.7) item( K, L ) :- atom( K, L ).  
       item( K, L ) :- list( K, L ).  
       atom( K, L ) :- letter( K, L ).  
       atom( K, M ) :- letter( K, L ), atom( L, M ).  
       letter( K, L ) :- terminal( a, K, L ).  
       .....  
       letter( K, L ) :- terminal( z, K, L ).  
       terminal( T, T.Y, Y ).
```





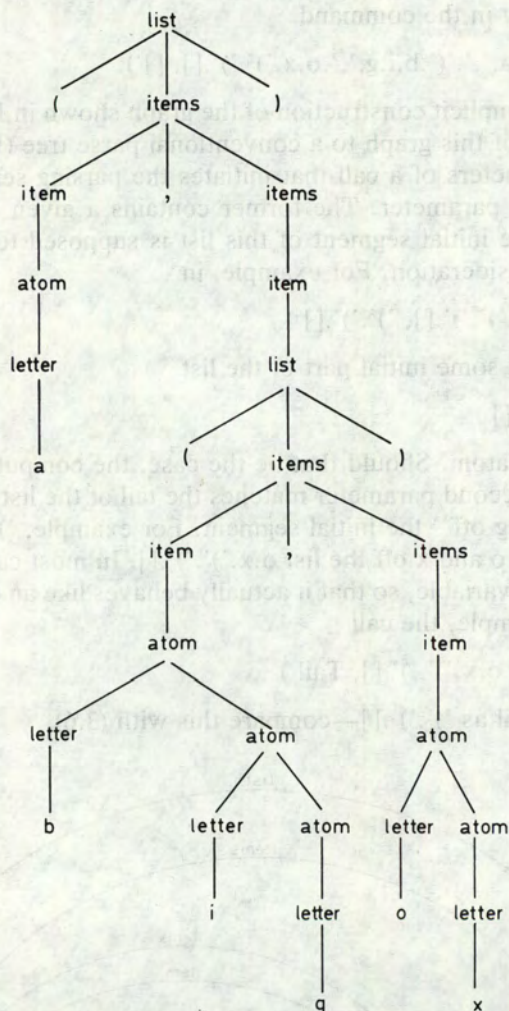


FIG. 3.3 The parse tree for the list (a,(big,ox)).

If the second parameter is a variable, only the entry node of some subgraph of the whole graph is known. Parsing then may give ambiguous results. For example, the call

```
items( b.i.g.',','.o.x.').').'.[], Tail )
```

might succeed with Tail instantiated to `',''.o.x.').').'.[]` or to `')).').').'.[]`. In general, the results depend on how the clauses of a parsing program are ordered. In the program above, the recursive clause for *items* would only

be activated because of forced failure coming after a successful parsing of *big* as *items*.

Recall now that the parameter of a Prolog procedure can, in principle, be bi-directional, the direction—input or output—depending on the form of the corresponding actual parameter. This also applies to calls that initiate parsing. If the first parameter is a variable, what we ask is whether there exists a sequence of terminal symbols that has a particular structure. For example, the call

```
list( AList, [] )
```

should instantiate *AList* to *any* valid list of terminal symbols; in other words, some list should be constructed, or synthesized. One example of such a list is the empty list.

However, the situation is not fully symmetric. For any given sequence of terminal symbols, a call on *list* either succeeds or fails, i.e. every sequence can be classified as a list or a non-list—can be syntactically analysed. Not so with synthesis. It is easy to see that the two calls

```
list( AList, [] ), fail
```

will act as a generator of one-element lists:

```
() (a) (b) ... (z) (aa) (ab) ... (az) (aaa) (aab) ...
```

Moreover, if we reorder the two clauses for *item*, the call on *item* with a variable first parameter would result in infinite recursion.

### 3.2. THE SIMPLEST FORM OF GRAMMAR RULES

The input and output parameters of the clauses that constitute a parsing program, such as (3.7), are the basis of yet another interpretation of those clauses: in terms of operations on sequences of terminal symbols. Take the clause

```
items( K, N ) :- item( K, L ), terminal( ',', L, M ), items( M, N ).
```

It can be read as follows: (an instance of) *items* can be “chopped off” (recognized at the beginning of) *K*, leaving *N*, if (an instance of) *item* can be chopped off *K*, leaving *L*, and then a comma can be chopped off *L*, leaving *M*, and finally (another instance of) *items* can be chopped off *M*, leaving *N*. Now the essence of all this is that *items* consist of an *item*, a comma, and *items*. The other information can be routinely added to this fundamental fact. All we need is four variables to stand for successive remainders of the initial sequence of terminal symbols.



In the notation we shall use henceforth, this routine information is suppressed. The notation resembles BNF productions. The lefthand side of a **Prolog grammar rule** names the construction, and the righthand side enumerates its constituents. For example:

atom  $\rightarrow$  letter, atom.

The symbol  $\rightarrow$  is rendered in Prolog as  $-->$  (it must be written without intervening blanks). There is a simple convention to distinguish nonterminal and terminal symbols: the latter are enclosed in square brackets, e.g.

items  $\rightarrow$  item, [ ' , ' ], items.

Contiguous terminal symbols can be enclosed in a single pair of brackets. For example, the rule for empty lists can be written as

list  $\rightarrow$  [ ' ( , ' ) ' ].

If all terminal symbols are characters (one-character nullary functors), we can use string notation:

list  $\rightarrow$  "( )".

Such grammar rules are merely syntactic sugar for the underlying clauses. The translation is fairly straightforward, the gain in clarity significant. However, some Prolog implementations, especially on small computers, do not support grammar rule notation. Even then it seems worthwhile to write a preprocessor in Prolog (we shall describe such a preprocessor in Section 7.4.4).

The counterpart of a parsing program, written down as a collection of grammar rules, will be called a **metamorphosis grammar**<sup>3</sup>, or grammar for short. Here is the grammar of lists, corresponding to the program (3.7).

```
list  $\rightarrow$  [ ' ( , ' ) ' ].
list  $\rightarrow$  [ ' ( ' ], items, [ ' ) ' ].
items  $\rightarrow$  item.
items  $\rightarrow$  item, [ ' , ' ], items.
item  $\rightarrow$  atom.
item  $\rightarrow$  list.
atom  $\rightarrow$  letter.
atom  $\rightarrow$  letter, atom.
letter  $\rightarrow$  [ a ].
.....
letter  $\rightarrow$  [ z ].
```

<sup>3</sup> This is the name invented by Colmerauer (1975, 1978). The name "definite clause grammars" was later introduced by Pereira and Warren (1980) for metamorphosis grammars in normal form (as defined by Colmerauer).



The procedure *terminal* need not be explicitly given (it ought to be provided by the implementation).

This grammar deserves its name. It is best understood independently of the Prolog program it has been used to conceal. Every rule reflects the "consist of" relationship between a whole and its constituents, exactly as the original BNF grammar does. However, it should be remembered that the grammar is also a program in disguise, and is executable *immediately*, without any additional effort on the programmer's part!

Parsing can be initiated in two ways. First, we can simply call one of the underlying procedures, e.g.

```
:- list( '('.a.', '.'('b.i.g.', '.o.x.').')'.[], [] ).
```

Second, we can use the built-in procedure *phrase* with two parameters: the nonterminal symbol and the sequence of terminal symbols (which is supposed to be an instance of the nonterminal). For example:

```
:- phrase( list, '('.a.', '.'('b.i.g.', '.o.x.').')'.[] ).
```

It should be pointed out that the first way brings out the routine information we just managed to hide. On the other hand, the second way is less flexible, e.g. we cannot use *phrase* to perform calls such as (3.8).

### 3.3. PARAMETERS OF NON-TERMINAL SYMBOLS

Grammars of the kind described so far are of little practical use. We seldom parse anything just to accept or reject it. More often than not, we need to compute the representation of its structure or to transform it somehow, and we must do this while accepting the input. The representation of the structure will be built step by step, with the terminal symbols taken into account in succession.

We shall give an example. Suppose we want to build a parse tree—a Prolog term—for every valid sequence of terminal symbols that constitute a list; see Fig. 3.3. To this end, we shall give each of the procedures in (3.7) an additional parameter to hold the representation (of a structure) to be constructed upon exit from the procedure. We must not meddle with input and output parameters: their role remains the same as before. Here is the program.

```
list( list( '(', ')' ), K, M ) :-  
    terminal( '(', K, L ), terminal( ')', L, M ).
```



```

list( list( '(', ITEMS, ')' ), K, N ) :-
    terminal( '(', K, L ), items( ITEMS, L, M ),
    terminal( ')', M, N ).
items( items( ITEM ), K, L ) :- item( ITEM, K, L ).
items( items( ITEM, ',', ITEMS ), K, N ) :-
    item( ITEM, K, L ), terminal( ',', L, M ),
    items( ITEMS, M, N ).
item( item( ATOM ), K, L ) :- atom( ATOM, K, L ).
item( item( LIST ), K, L ) :- list( LIST, K, L ).
atom( atom( LETTER ), K, L ) :- letter( LETTER, K, L ).
atom( atom( LETTER, ATOM ), K, M ) :-
    letter( LETTER, K, L ), atom( ATOM, L, M ).
letter( letter( a ), K, L ) :- terminal( a, K, L ).
.....
letter( letter( z ), K, L ) :- terminal( z, K, L ).

```

Again, we shall suppress the routine information, i.e. leave out the input and output parameters. The resulting grammar will be as follows:

```

list( list( '(', ')' ) ) → [ '(', ')' ].
list( list( '(', ITEMS, ')' ) ) →
    [ '(', items( ITEMS ), [ ')' ] ].
items( items( ITEM ) ) → item( ITEM ).
items( items( ITEM, ',', ITEMS ) ) →
    item( ITEM ), [ ',', items( ITEMS ) ].
item( item( ATOM ) ) → atom( ATOM ).
item( item( LIST ) ) → list( LIST ).
atom( atom( LETTER ) ) → letter( LETTER ).
atom( atom( LETTER, ATOM ) ) → letter( LETTER ),
    atom( ATOM ).
letter( letter( a ) ) → [ a ].
.....
letter( letter( z ) ) → [ z ].

```

To compute the parse tree of Fig. 3.3, call:

```
:- phrase( list( T ), '(.a.,'.('b.i.g.,'.o.x.)')'.[] ).
```

The conciseness and power of metamorphosis grammars can hardly be appreciated in this tiny example. We shall show a grammar that describes (and parses) sequences of statements of a simple programming language. The admissible statements are: assignment, if-then-else-fi,

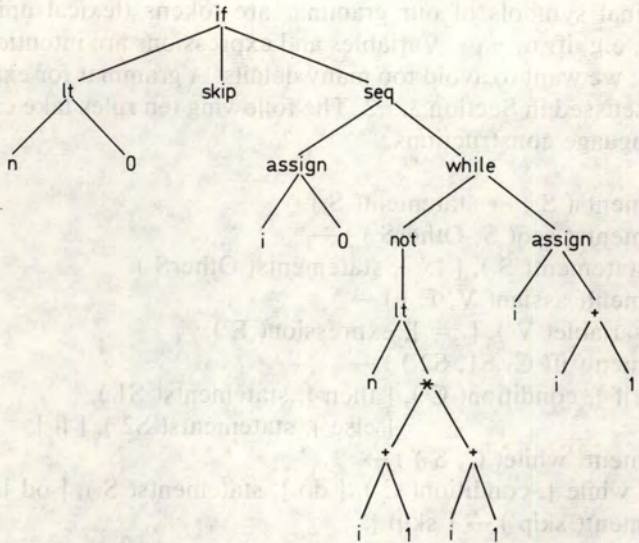


FIG. 3.4 An abstract syntax tree.

while-do-od, and skip. The sequencing operator is the semicolon. The condition is either an arithmetic relation ( $=$  or  $<$ ) or a relation negated<sup>4</sup>.

The intended meaning of a sequence of statements is the term that shows its structure. We shall not go into details; instead, we shall give an example which ought to explain the idea. Given the (one-element) sequence of statements:

```

if  $n < 0$  then skip else
   $i := 0$  ;
  while not  $n < (i + 1) * (i + 1)$  do
     $i := i + 1$ 
  od
fi

```

we should obtain the abstract syntax tree (a Prolog term):

(3.9)  $\text{if}(\text{lt}(n, 0), \text{skip}, \text{seq}(\text{assign}(i, 0),$   
 $\text{while}(\text{not}(\text{lt}(n, \text{'*'}(\text{'+'}(i, 1), \text{'+'}(i, 1)))),$   
 $\text{assign}(i, \text{'+'}(i, 1)))))$

The same tree is shown in Fig. 3.4.

<sup>4</sup> Both parts of this example, here and in Section 3.4.1, are modelled on the illustration in Colmerauer's original paper (1975).



Terminal symbols of our grammar are tokens (lexical units of the language), e.g. if, n, +, (. Variables and expressions are intentionally left undefined: we want to avoid too many details. A grammar for expressions will be discussed in Section 3.5.2. The following ten rules take care of the rest of language constructions.

```

statements( S ) → statement( S ).
statements( seq( S, OtherS ) ) →
    statement( S ), [ ';' ], statements( OtherS ).
statement( assign( V, E ) ) →
    variable( V ), [ ':=' ], expression( E ).
statement( if( C, S1, S2 ) ) →
    [ if ], condition( C ), [ then ], statements( S1 ),
    [ else ], statements( S2 ), [ fi ].
statement( while( C, S ) ) →
    [ while ], condition( C ), [ do ], statements( S ), [ od ].
statement( skip ) → [ skip ].
condition( R ) → relation( R ).
condition( not( R ) ) → [ 'not' ], relation( R ).
relation( eq( E1, E2 ) ) →
    expression( E1 ), [ '=' ], expression( E2 ).
relation( lt( E1, E2 ) ) →
    expression( E1 ), [ '<' ], expression( E2 ).

```

This grammar would probably be activated by calls such as

```

... read_a_list_of_tokens( LisT ),
    phrase( statements( Structure ), LisT ) ...

```

which analyse LisT and instantiate Structure appropriately, or fail if LisT is not a valid sequence of statements. Another possibility (not always practical, though) is to build—synthesize, if you prefer—a list of Tokens starting from a given structure:

```

... take_a_structure( S ), phrase( statements( S ), Tokens ) ...

```

Here, Tokens will be instantiated if only S is a proper structure. The grammar establishes one-one correspondence between structures and lists of tokens, and provides transformation both ways.

A more realistic example of synthesis based on a metamorphosis grammar will be given in the next section. Here we only observe that in both cases (analysis and synthesis) similar computations ensue. They differ because, on analysis, the sequence of terminal symbols “controls”

the computation (i.e. determines the choice of rules) whereas, on synthesis, it is "controlled" by the initial non-terminal symbol's parameter.

### 3.4. EXTENSIONS

#### 3.4.1. Conditions

Grammar rules described so far correspond to clauses in which every call manipulates the sequence of terminal symbols, i.e. every call has an input and an output parameter. Other calls could be inserted in between without affecting the transfer of terminal symbols. The question is: Would it be useful, and how could it be interpreted?

As a simple possibility, consider the cut in the first clause of *list*:

```
list( list( '(', ')'), K, M ) :-
    terminal( '(', K, L ), terminal( ')', L, M ), !.
```

The cut turns the computation based on the *list* procedure into a "deterministic" process: it handles either the empty list or non-empty lists. It does not matter when we want to recognize a list. However, it is now impossible to generate lists. The command

```
:- list( L, T, [] ), write( L ), write( T ), nL, fail.
```

will only write one instance of L and T, namely

```
list( '(', ')') and '().'.[]
```

The gain from the cut is small in this case, anyway. Cuts would be of much greater use, say, in the program that parses statements (see the previous section), where long and deep computations may occur.

Another example: suppose we want to change the program for parsing lists so that for an atom it produces a Prolog atom instead of a parse tree, e.g. returns

```
list( '(', items( item( big ), ',', items( item( ox ) ) ), ')')
```

for the list (big,ox). One way to do so is to make the procedure for atoms return a Prolog list of letters, and apply the built-in procedure *pname* (see Section 5.10) to this list

```
item( item( ATOM ), K, L ) :-
    atom( LETTERS, K, L ), pname( ATOM, LETTERS ).
item( item( LIST ), K, L ) :- list( LIST, K, L ).
atom( LETTER.[], K, L ) :- letter( LETTER, K, L ).
```



```

atom( LETTER.LETTERS, K, M ) :-
    letter( LETTER, K, L ), atom( LETTERS, L, M ).
letter( a, K, L ) :- terminal( a, K, L ).
.....
letter( z, K, L ) :- terminal( z, K, L ).

```

One final example: in the program above we shall replace the 26 clauses that define letters by a single clause:

```

letter( LETTER, K, L ) :-
    terminal( LETTER, K, L ), isletter( LETTER ).

```

with *isletter* defined, say, as

```

isletter( LETT ) :- a @=< LETT, LETT @=< z.

```

This new clause can be used as follows:

```

letter( Lett, x.''.').[], Tail ).
terminal( Lett, x.''.').[], Tail ), isletter( Lett ).
    Lett ← x,      Tail ← ''.').[]
isletter( x ).
etc.

```

The variable in the call on *terminal* matches *every* terminal symbol. If the terminal symbol is *not* a letter, a call on *isletter* will fail and a letter will not be recognized. We call such terminal symbols **variable terminals**: the first (still unprocessed) symbol is selected and is then either accepted or rejected, e.g. according to the result of a test such as *isletter*.

Extra calls that do not comprise input and output parameters have been known as *conditions*, but the name is slightly misleading. Only in the last example *isletter(Lett)* can be interpreted as a condition: the clause will only be applied if *isletter* succeeds. The call on *pname* in the second example is rather an action performed on the parameters of non-terminal symbols. Finally, the cut can be reasonably interpreted exactly as in any other clause, as pragmatic information on the future use of the clause.

Conditions in metamorphosis grammars are enclosed in curly brackets, so that they will not be confused with terminal and non-terminal symbols. Examples:

```

list( list( '(', ')' ) ) → [ '(', ')' ], {!}.
item( item( ATOM ) ) → atom( LETTERS ),
    { pname( ATOM, LETTERS ) }.
letter( LETTER ) → [ LETTER ], { isletter( LETTER ) }.

```

As an exception, the cut need not be placed within curly brackets, e.g.

```

list( list( '(', ')' ) ) → [ '(', ')' ], !.

```

Contiguous conditions can be combined in a single pair of brackets, and in general a condition can also contain alternatives conjoined by semicolons, e.g.

$\text{alphanum}(\text{Char}) \rightarrow [\text{Char}], \{ \text{isletter}(\text{Char}) ; \text{isdigit}(\text{Char}) \}.$

We shall now present a small fragment of a metamorphosis grammar, meant primarily for synthesis (but applicable both ways, although not without reservations). We want to take a structure computed by the grammar for statements (see the previous section) and produce its translation into a machine-oriented symbolic language. We shall only give a hint of the target language by showing schematic translations of  $\text{while}(C, S)$  and  $\text{if}(C, S_1, S_2)$ .

Let  $\bar{C}$  and  $\bar{S}$  be the translations of  $C$  and  $S$ . The evaluation of  $\bar{C}$  sets a flag used implicitly by a conditional jump instruction. Let  $\ell_1, \ell_2$  be unique labels. The translation of  $\text{while}(C, S)$  will be

$\text{label}(\ell_1)$   
 $\text{not}(\bar{C})$   
 $\text{jumpiftrue}(\ell_2)$   
 $\bar{S}$   
 $\text{jump}(\ell_1)$   
 $\text{label}(\ell_2)$

The translation of  $\text{if}(C, S_1, S_2)$  will be

$\bar{C}$   
 $\text{jumpiftrue}(\ell_1)$   
 $\bar{S}_2$   
 $\text{jump}(\ell_2)$   
 $\text{label}(\ell_1)$   
 $\bar{S}_1$   
 $\text{label}(\ell_2)$

The “code generator” can be written as a grammar of the target language. By way of explanation, we shall show three of the rules that belong to the uppermost level of the definition:

$\text{code}(\text{seq}(S, \text{OtherS})) \rightarrow \text{code}(S), \text{code}(\text{OtherS}).$   
 $\text{code}(\text{while}(C, S)) \rightarrow$   
 $\quad \{ \text{newlabel}(L_1) \}, [\text{label}(L_1)], \text{codecond}(\text{not}(C)),$   
 $\quad \{ \text{newlabel}(L_2) \}, [\text{jumpiftrue}(L_2)], \text{code}(S),$   
 $\quad [\text{jump}(L_1), \text{label}(L_2)].$   
 $\text{code}(\text{skip}) \rightarrow [].$

The action *newlabel* can generate a new, unique label. The definition of *codecond* will be given below. The third rule illustrates a new feature of



grammar rules. If the righthand side contains no terminal and non-terminal symbols, nothing will be produced during synthesis and nothing will be “chopped off” during analysis. The underlying clause is

```
code( skip, K, K ).
```

Try to trace the execution of

```
:- code( seq( skip, skip ), Translation, [ ] ).
```

Assuming that *codere1* defines the grammar of codes for relations *eq* and *lt*, the definition of *codecond* can be as follows:

```
codecond( not( not( C ) ) ) → codecond( C ).
codecond( not( Rel ) ) → codere1( Rel ), [ revert( _ ) ].
codecond( Rel ) → codere1( Rel ).
```

where “revert” is an instruction of the target language that resets the “condition flag”.

The example would be completed after specifying the translation of expressions and of assignments, in particular the handling of variables.

The code generator together with the grammar of statements might constitute the core of a simple compiler. Its overall structure might be:

```
compile :- read_tokens( Token_list ),
           parse( Token_list, Syntax_tree ),
           generate_code( Syntax_tree, Object_code ),
           write_code( Object_code ).
```

with *parse* and *generate\_code* defined as

```
parse( T, S ) :- phrase( statements( S ), T ).
generate_code( S, O ) :- phrase( code( S ), O ).
```

The procedure *read\_tokens*, reading the source program in and performing lexical analysis, might also be (partly) written as a metamorphosis grammar—see Colmerauer (1975, 1978).

### 3.4.2. Context

Another feature of grammar rules in Prolog is a mechanism for modifying the sequence of terminal symbols during the computation. In general, this would require explicit manipulations on input and output parameters, but such general mechanisms seem only necessary in natural language processing (an important application of Prolog). A very restricted mechanism, so-called context grammar rules, is quite sufficient, though, in most of the other applications.



In a context grammar rule, the lefthand side is supplemented by a so-called **context**<sup>5</sup>: terminal symbols, preceding the arrow  $\rightarrow$ . For example:

```
otherst( S, S ), [ Delim ]  $\rightarrow$  [ Delim ], { stsdelim( Delim ) }.
do, [ 'not' ]  $\rightarrow$  dont.
```

The output parameter in the head of an underlying clause is appended to the context. As clauses, the above rules are:

```
otherst( S, S, K, Delim.L ) :-
    terminal( Delim, K, L ), stsdelim( Delim ).
do( K, 'not'.L ) :- dont( K, L ).
```

The first rule can be interpreted without resorting to the corresponding clause; we shall give the interpretation below. The second rule, however, can only be explained in terms of manipulations on sequences of terminal symbols: a *new* terminal symbol appears after recognizing an instance of *dont*, and only then is an instance of *do* recognized as well. We shall elaborate on this example a little, too.

First we come back to the grammar for statements. In its present shape it performs rather poorly on incorrect inputs. It fails without giving any message or diagnostics. We shall try to improve the definition of *statements*, leaving the other rules as an exercise. We observe that a statement (other than the last) may be delimited by a semicolon (it indicates that there are other statements in this sequence), by **else**, **fi**, or **od**. Other delimiters are erroneous. In case of errors, no meaningful structure may be found for the whole sequence of statements, but we elect to continue the analysis, after skipping a portion of input up to the nearest semicolon. Here are some rules of a grammar that implements these ideas.

```
statements( Sts )  $\rightarrow$  statement( St ), otherst( St, Sts ).
otherst( St1, seq( St1, Sts ) )  $\rightarrow$ 
    [ ';' ], statement( St2 ), otherst( St2, Sts ).
otherst( St, St ), [ Delim ]  $\rightarrow$ 
    [ Delim ], { stsdelim( Delim ) }.
otherst( -, - )  $\rightarrow$  [ T ], erroneous( T ).
otherst( St, St )  $\rightarrow$  []. % this for the last statement
erroneous( T )  $\rightarrow$  { write( bad( T ) ), nl }, skipped.
skipped, [ ';' ]  $\rightarrow$  [ ';' ].
```

<sup>5</sup> Readers familiar with context-sensitive grammars will notice that neither rule is a proper context-sensitive rule. Even if we disregard parameters and conditions, the rules will only belong to Chomskian type 0.



skipped  $\rightarrow$  [ \_ ], skipped.

skipped  $\rightarrow$  []. % if we are skipping the last statement

stdelim( else ). stdelim( fi ). stdelim( od ).

The context rule can be interpreted in the following manner: "the remainder of a sequence of statements is empty if we have encountered a proper delimiter; this delimiter is retained". Notice that we have actually effected one-item lookahead on a list of terminal symbols. In general, we can have lookahead for any *fixed* number of terminal symbols, for example

$p, [T1, T2] \rightarrow [T1, T2], \{ \text{test}(T1, T2) \}.$

This translates into

$p(K, T1.T2.M) :-$

terminal( $T1, K, L$ ), terminal( $T2, L, M$ ), test( $T1, T2$ ).

We can use  $p$  to make the *test*; e.g. in

$a \rightarrow p, b, c.$

$p$  consumes no input, so that the rule is structurally equivalent to

$a \rightarrow b, c.$

but it will only be applied if two leftmost terminal symbols of the current sequence pass the test.

The second example is a very simplified little grammar that recognizes auxiliary "do not", "don't", "does not", "doesn't". This particular problem can easily be solved differently; the way we have chosen is intended as an illustration of context grammar rules:

$\text{aux} \rightarrow \text{do}, [\text{'not'}].$

$\text{do}, [\text{'not'}] \rightarrow \text{dont}.$

$\text{do} \rightarrow [\text{do}].$

$\text{do} \rightarrow [\text{does}].$

$\text{dont} \rightarrow [\text{'don't'}].$  %i.e. don't

$\text{dont} \rightarrow [\text{'doesn't'}].$  %i.e. doesn't

The following computation should explain how this grammar is used:

$\text{aux}(\text{'doesn't'}. \text{like.it}([], \text{Tail})).$

$\text{do}(\text{'doesn't'}. \text{like.it}([], T1), \text{terminal}(\text{'not'}, T1, \text{Tail})).$

$T1 \leftarrow \text{'not'}.L$

$\text{dont}(\text{'doesn't'}. \text{like.it}([], L),$

$\text{terminal}(\text{'not'}, \text{'not'}.L, \text{Tail})).$

```

terminal( 'doesn't', 'doesn't'.like.it.[], L ),
    terminal( 'not', 'not'.L, Tail ).
    L ← like.it.[ ]
terminal( 'not', 'not'.like.it.[], Tail ).
    Tail ← like.it.[ ]
success

```

Our last example is a small grammar that discards leading zeroes from an integer represented as a list of digits:

```

zeroes, [ D ] → [ 0 ], zeroes, [ D ], { digit( D ) }.
zeroes → [ ].

```

You may wish to trace the execution of the directives

```

:- zeroes( 0.3.[ ], Tail ).
:- zeroes( 0.0.[ ], Tail ).

```

### 3.4.3. Alternatives

Two or more grammar rules with the same lefthand side (including context and parameters of the non-terminal symbol) can be combined into a single rule with the common lefthand side and with the righthand side taking the form of **alternatives**—a sequence of original righthand sides separated by semicolons. For example:

```

list → [ '(', ')' ] ; [ '(', ], items, [ ')' ].
items → item ; item, [ ',', ], items.
item → atom ; list.
atom → letter ; letter, atom.
letter → [ L ], { isletter( L ) }.

```

Notice how—at last—we managed to come back rather closely to the original BNF grammar (3.1).

The translation of a rule with an alternative into an underlying clause is straightforward. One example should be sufficient:

```

items( K, N ) :- item( K, N ) ;
                item( K, L ), terminal( ',', L, M ), items( M, N ).

```

The notation with alternatives is, strictly speaking, a “convenience” rather than a real extension, and—like alternatives in ordinary clauses (see Section 1.3.7)—it can sometimes adversely affect the grammar’s readability.



### 3.4.4. Syntax of Grammar Rules: Summary

We shall now give a metamorphosis grammar that describes full syntax of grammar rules supported by Prolog-10. The principles of mapping rules onto underlying clauses have been discussed at length in the previous sections, so we choose not to overburden the grammar with parameters that would take care of the translation. However, we encourage you to try and augment the grammar along these lines. A hint: most of the non-terminal symbols should be given three parameters, two variables (to construct an input and output parameter) and a term (to hold the—partial—translation). For example:

```
grammar_rule( ( Tr_of_left :- Tr_of_right ), In_var, Out_var )
    → lefthand_side( Tr_of_left, In_var, Out_var ), [ '→' ],
       righthand_side( Tr_of_right, In_var, Out_var ), [ '.' ].
rule_items( ( Tr_of_item, Tr_of_items ), Curr_in_var, Out_var )
    → rule_item( Tr_of_item, Curr_in_var, Mid_var ), [ ',' ],
       rule_items( Tr_of_items, Mid_var, Out_var ).
```

In the actual translation we might eliminate the calls on the procedure *terminal*. Since *terminal*(*T*, *K*, *L*) means that  $K = T.L$ , we can substitute in advance  $T.L$  for *K* elsewhere in the clause. For example, in the clause

```
list( K, N ) :-
    terminal( '(', K, L ), items( L, M ), terminal( ')', M, N ).
```

we have  $K = '(.L$  and  $M = ')'.N$ , and after replacing *K* and *M* we obtain

```
list( '(.L, N ) :- items( L, ')'.N ).
```

This is, in fact, what is done in many implementations (see, e.g., Section 7.4.9). As we have executed both calls on *terminal* beforehand, every computation started by a call on *list* will be at least two steps shorter. Here are some other examples of such an improved translation of grammar rules:

```
letter( Lett, Lett.L, L ) :- isletter( Lett ).
p( T1.T2.M, T1.T2.M ) :- test( T1, T2 ).
zeroes( 0.L, D.N ) :- zeroes( L, D.N ), digit( D ).
```

We shall now present the grammar *without* parameters (it is, really, equivalent to a BNF definition).

```
grammar_rule → lefthand_side, [ '→' ],
               righthand_side, [ '.' ].
lefthand_side → nonterminal, context.
context → terminals ; [].
```

```

righthand_side → alternatives.
alternatives → alternative ;
                alternative, [ ';' ], alternatives.
alternative → [ [] ] ; rule_items.
rule_items → rule_item ; rule_item, [ ';' ], rule_items.
rule_item → nonterminal ; terminals ; condition ; [ ! ] ;
            [ '(' ], alternatives, [ ')' ].
nonterminal → name ;
              name, [ '(' ], list_of_terms, [ ')' ].
terminals → [ '[' ], list_of_terms, [ ']' ] ; string.
condition → [ '{' ], procedure_body, [ '}' ].
list_of_terms → term ; term, [ ';' ], list_of_terms.

```

Definitions of name, term, string and procedure\_body are left as an exercise.

It should be noted that the original appearance of grammar rules in the Marseilles interpreter of Prolog I (Roussel 1975) was slightly different. In particular, no alternatives were allowed, and terminal symbols and conditions could not be combined. Just to give the flavour of it, we shall rewrite in Marseilles syntax some of the grammar rules for statements (Section 3.4.2).

```

:STATEMENTS( *STS ) == :STATEMENT( *ST )
                        :OTHERST( *ST, *STS ).
:OTHERST( *ST1, SEQ( *ST1, *STS ) ) ==
#; :STATEMENT( *ST2 ) :OTHERST( *ST2, *STS ).
:OTHERST( *ST, *ST ) ##*DELIM ==
##*DELIM -STSDELIM( *DELIM ).
:OTHERST( *DUMMY1, *DUMMY2 ) ==
##*T :ERRONEOUS( *T ).
:OTHERST( *ST, *ST ) == .
      *THIS FOR THE LAST STATEMENT.

```

### 3.5. PROGRAMMING HINTS

#### 3.5.1. Efficiency Considerations

Metamorphosis grammars correspond to Prolog programs which implement a very general parsing strategy: nondeterministic top-down parsing with backtracking (Aho and Ullman 1977; Gries 1971). The potential cost of this strategy is exponential. This is the disadvantage of the gener-



ality and ease of programming with metamorphosis grammars. Well-known parsing algorithms for restricted classes of context-free grammars can be quite conveniently programmed in Prolog without metamorphosis grammars. See for example the operator precedence parser described in Section 7.4.3 and Appendix A.3. However, this requires explicit handling of the parsing stack, attributes etc., while metamorphosis grammars by themselves are as powerful as attribute grammars (Knuth 1968) or two-level grammars (van Wijngaarden 1976)—see the discussion in (Pereira and Warren 1980). Parameters and conditions/actions make it possible to construct an intuitively appealing, concise and readable metamorphosis grammar of any existing programming language (and of reasonable subsets of natural languages), capturing semantics as well as syntax—see e.g. (Moss 1979). At the same time, such a grammar can usually be used as a translator of this language, without additional effort on the part of the programmer, but there is often a certain price to be paid in efficiency.

One source of inefficiency is repetition. Consider two rules from the grammar for statements (Section 3.3):

```
relation( eq( E1, E2 ) ) →
    expression( E1 ), [ '=' ], expression( E2 ).
relation( lt( E1, E2 ) ) →
    expression( E1 ), [ '<' ], expression( E2 ).
```

If a given relation is not an equality, we recognize this state of affairs only after parsing the first expression and failing to find an equals sign. We abandon the rule and choose the next but then we must once more parse the first expression (which may be quite large). The problem remains if we change the order of the rules.

To avoid this inefficiency, we may apply **factorization**—the technique already used in Section 3.4.2:

```
relation( R ) → expression( E1 ), op_and_expr( E1, R ).
op_and_expr( E1, eq( E1, E2 ) ) → [ '=' ], expression( E2 ).
op_and_expr( E1, lt( E1, E2 ) ) → [ '<' ], expression( E2 ).
```

Another solution is to combine the original rules into a single rule by replacing the terminal symbols with a variable terminal, and adding a suitable condition:

```
relation( R ) → expression( E1 ), [ Op ],
    { makestruct( Op, E1, E2, R ) },
    expression( E2 ).
makestruct( '=', E1, E2, eq( E1, E2 ) ).
makestruct( '<', E1, E2, lt( E1, E2 ) ).
```



Notice the position of the condition: if we placed it at the end of the rule, we would run the risk of discovering an improper instance of Op only after parsing the whole input, say,

$$(A + b/2) * c \text{ blah\_blah } 2 * (n - (x + y) / 4)$$

In its present position the condition fails as soon as it sees an invalid operator.

Both improvements of the original grammar eliminate possible repetitions. Both, though, seem to decrease the readability and elegance of the original solution, and we recommend that they be applied (if at all necessary) only in the late stages of program debugging.

### 3.5.2. Elimination of Left Recursion

We shall now discuss a problem which frequently arises with inexperienced use of metamorphosis grammars. As an example, we shall consider the task of writing a workable grammar of simple arithmetic expressions (see Section 3.3). Here is the definition in BNF (for simplicity, we limit ourselves to two operators only):

$$\begin{aligned} \langle \text{expression} \rangle &::= \langle \text{add\_expr} \rangle \mid \\ &\quad \langle \text{expression} \rangle + \langle \text{add\_expr} \rangle \mid \\ &\quad \langle \text{expression} \rangle - \langle \text{add\_expr} \rangle \\ \langle \text{add\_expr} \rangle &::= \langle \text{constant} \rangle \end{aligned}$$

We now give an obvious transcription of this definition into a metamorphosis grammar. Parameters are used to build the structure of a given expression—see (3.9).

$$\begin{aligned} \text{expression}(E) &\rightarrow \text{add\_expr}(E). \\ \text{expression}(E1 + E2) &\rightarrow \\ &\quad \text{expression}(E1), [' + '], \text{add\_expr}(E2). \\ \text{expression}(E1 - E2) &\rightarrow \\ &\quad \text{expression}(E1), [' - '], \text{add\_expr}(E2). \end{aligned}$$

The definition of *add\_expr* will be left out (it can be simply an integer constant).

Unfortunately, this grammar—as a program—is not only inefficient but also incorrect. It goes into infinite (left) recursion whenever we give it an expression that contains a minus. Try to analyse the expression  $2 - 3 + 5$  (represented by 2.'-' .3.'+' .5.[]).



At first sight, it seems we can improve the situation by applying one of the techniques shown in the previous section. For example, the second technique gives the following rules:

```

expression( E ) → add_expr( E ).
expression( E ) → expression( E1 ), [ Op ],
                        { makesum( Op, E1, E2, E ) },
                        add_expr( E2 ).
makesum( '+', E1, E2, E1 + E2 ).
makesum( '-', E1, E2, E1 - E2 ).

```

Now correct expressions will be parsed successfully, although an expression composed of  $n$  add-expressions will require  $n - 1$  backtracks before reaching the solution. But the grammar will still fall into infinite recursion on any incorrect input (you may wish to check this on  $2 + .[]$ ). This means that it is of no practical value. As in all top-down parsing methods, we must eliminate left recursion to avoid trouble.

Suppose we reverse nonterminal symbols in the recursive rules in (3.10):

```

expression( E1 + E2 ) → add_expr( E1 ), [ '+' ], expression( E2 ).
expression( E1 - E2 ) → add_expr( E1 ), [ '-' ], expression( E2 ).

```

Now incorrect input causes the grammar to fail (without any error message, but this can be fixed). However, this grammar interprets operators as right-associative. The instantiation of its parameter for the expression  $2 - 3 + 5$  will be  $-(2, +(3, 5))$  rather than  $+(-(2, 3), 5)$ . Here is a possible solution to this new problem:

```

expression( E ) → add_expr( E1 ), rest_of_expression( E1, E ).
rest_of_expression( E1, E ) →
    [ '+' ], add_expr( E2 ), rest_of_expression( E1 + E2, E ).
rest_of_expression( E1, E ) →
    [ '-' ], add_expr( E2 ), rest_of_expression( E1 - E2, E ).
rest_of_expression( E1, E1 ) → [].

```

When we parse an expression, the parameter is initially uninstantiated. It is passed unchanged and instantiated after reaching the end of the expression. (In the terminology of attribute grammars this is a synthesized attribute.) The final structure is accumulated step by step. For example, during the parsing of the expression  $2 - 3 + 4 - 5$ , *rest\_of\_expression* will be activated four times, with  $2$ ,  $2 - 3$ ,  $(2 - 3) + 4$  and  $((2 - 3) + 4) - 5$  as the first parameter. (This parameter is an inherited attribute.) Eventually the third rule will be chosen and  $E$  instantiated to  $((2 - 3) + 4) - 5$ .



We shall now present a grammar for expressions, complete with error handling, that fits the grammar for statements (see Sections 3.3 and 3.4.2). The definition of *erroneous* was given in Section 3.4.2.

```

expression( E ) → add_expr( E1 ), rest_of_expression( E1, E ).
rest_of_expression( E1, E ) →
    [ '+' ], add_expr( E2 ), rest_of_expression( E1 + E2, E ).
rest_of_expression( E1, E ) →
    [ '-' ], add_expr( E2 ), rest_of_expression( E1 - E2, E ).
rest_of_expression( E1, E1 ), [ Termin ] →
    [ Termin ], { expr_termin( Termin ) }.
rest_of_expression( _, _ ) → [ T ], erroneous( T ).
rest_of_expression( E1, E1 ) → [].
expr_termin( then ).          expr_termin( else ).
expr_termin( do ).           expr_termin( od ).
expr_termin( ';' ).          expr_termin( fi ).
add_expr( E ) → mult_expr( E1 ), rest_of_add_expr( E1, E ).
rest_of_add_expr( E1, E ) →
    [ '*' ], mult_expr( E2 ), rest_of_add_expr( E1 * E2, E ).
rest_of_add_expr( E1, E ) →
    [ '/' ], mult_expr( E2 ), rest_of_add_expr( E1 / E2, E ).
rest_of_add_expr( E1, E1 ), [ Termin ] →
    [ Termin ], { add_expr_termin( Termin ) }.
rest_of_add_expr( _, _ ) → [ T ], erroneous( T ).
rest_of_add_expr( E1, E1 ) → [].
add_expr_termin( Termin ) :- expr_termin( Termin ).
add_expr_termin( '+' ).
add_expr_termin( '-' ).
mult_expr( E ) → variable( E ).
mult_expr( E ) → constant( E ).
mult_expr( E ) → [ '(' ], expression( E ), [ ')' ].

```

To make the grammar really complete, we should also define variables and constants. We choose not to do it, because variables require symbol table handling—we shall discuss it in Section 4.2.2.

The techniques described above are only necessary if we want to perform analysis with a metamorphosis grammar. Even more: the transformed grammar is not good for synthesis, i.e. for constructing the sequence of terminal symbols given a (correct!) structure. Specifically, for synthesizing expressions, the only reasonable solution would be the original grammar (3.10).



---

## 4 SIMPLE PROGRAMMING TECHNIQUES

---

### 4.1. INTRODUCTION

Programming in Prolog differs from programming in classical (Pascal-style) languages primarily at the level of individual procedures. The larger the program, the more suitable the general recommendations of programming methodology. The advantages of systematic top-down design of programs, modularity<sup>1</sup>, clean interfaces, etc., are certainly independent of the programming language used. Design and coding techniques specific to Prolog are due to its *logical* origin.

In Section 1.3.4 and Chapter 2 we discussed logical—static—interpretation of procedures. This interpretation makes it possible to design programs without paying attention—at least initially—to *how* the computation will proceed. One only needs to indicate *what* will be computed. Kowalski (1974, 1979a) coined an “equation”,

Algorithm = Logic + Control

which helps clarify the distinctive feature of logic programming. It is maintained that logic programming relieves the programmer of the burden of specifying control information for her program. One would like to say: completely relieves, but unfortunately (at least in Prolog) this is not the case. Many useful built-in procedures, such as the cut, input/output and program modification procedures (assert, etc.; see Section 5.11), cannot be interpreted statically. As a result, a *practical* program may not usually be designed without paying regard to control information.

<sup>1</sup> At least on a conceptual level: most existing Prolog implementations do not support it explicitly.

In Section 4.3 we shall briefly consider the advantages and disadvantages of some side-effects in Prolog; we shall also present several simple tricks that help increase the efficiency of Prolog programs (especially their space requirements) in many existing implementations. Earlier, in Section 4.2, we shall give a few examples of Prolog implementation of commonly used data structures, in particular binary trees and linear lists. We shall show basic operations on those structures and a few typical applications. Section 4.4 contains small examples of program design.

## 4.2. EXAMPLES OF DATA STRUCTURES

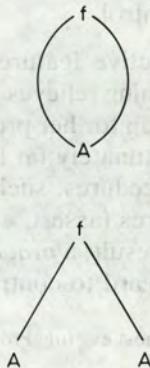
We have chosen unbalanced binary search trees (BSTs) and one-way linear lists as an illustration of methods of implementing recursive data structures in Prolog. We assume you are familiar with basic definitions and algorithms; a detailed, though rather elementary presentation can be found, for example, in Wirth (1976) or Sedgewick (1983). Here, we shall refer only to common intuitions, and we shall concentrate on problems specific to Prolog.

We shall also briefly discuss representation of data structures by clauses—in particular, Prolog counterparts of arrays.

### 4.2.1. Simple Trees and Lists

Terms can usually be regarded as trees: the main functor labels the root, subtrees correspond to arguments. This is slightly imprecise, because multiple occurrences of variables represent more general structures—directed acyclic graphs (DAGs). However, the term  $f(A, A)$  which should be depicted as

can be thought of as





We must only remember that the two subtrees will remain identical, so instantiating variables in one will affect the other. Another difficulty is that it is possible to compute terms which are not even DAGs, and which should therefore be regarded as corresponding to infinite trees (see Section 1.2.3). All the same, an ordinary tree is a good intuition of the (general) term.

Terms are a convenient and concise representation of trees with irregular structure, where the information in the nodes determines both the shape of the tree and the repertory of applicable operations. The abstract syntax tree of Fig. 3.3, Section 3.3, is a typical example. However, programs that manipulate such irregular structures are usually problem-dependent, in that every principal functor (i.e. every type of node may require different computations).

There are other situations, typified by binary search trees, when we need a more uniform representation, because we use trees for contents rather than for structure. Suppose we represent the BST of Fig. 4.1 as the term

```
few(people(many(languages), speak))
```

Even if we disregard the ambiguity (is “languages” the left or right descendant of “many”?), main functors and their arguments must be isolated, that is, we must use the built-in procedure =.. (“univ”; see Section 5.10). To modify the tree, e.g. by adding a node, we must rebuild it completely, also using *univ*. This is not only inelegant, but inefficient as well (but see Section 4.2.6 for a discussion of such techniques).

We shall therefore represent empty binary trees by the atom

```
nil
```

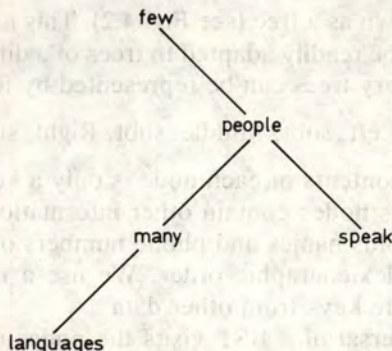


FIG. 4.1 A binary search tree.

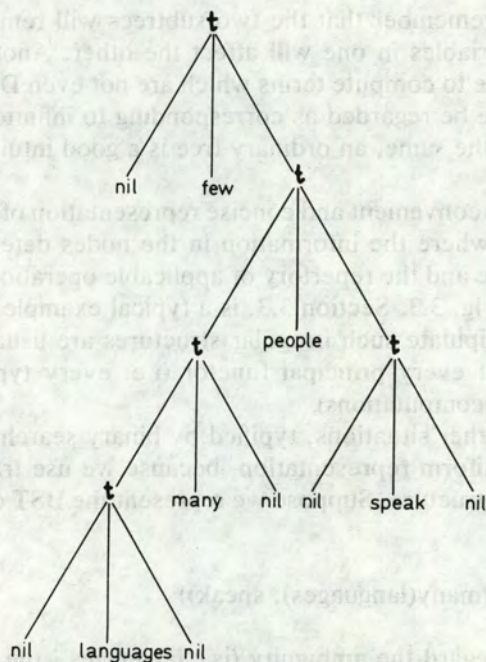


FIG. 4.2 A representation of the tree of Fig. 4.1.

and nonempty trees by three-argument terms

`t( Left_subtree, Node_info, Right_subtree )`

For example, the BST of Fig. 4.1 will be represented by the term

`t( nil, few, t( t( t( nil, languages, nil ), many, nil ),  
people, t( nil, speak, nil ) ) )`

The term can be drawn as a tree (see Fig. 4.2). This method of representing binary trees can be readily adapted to trees of a different fixed degree, e.g. non-empty ternary trees can be represented by four-argument terms

`tt( Node_info, Left_subt, Middle_subt, Right_subt )`

In Fig. 4.2 the contents of each node is only a key, but of course in practical applications nodes contain other information as well. The tree shown in Fig. 4.3 holds names and phone numbers of several persons—names are keys in lexicographic order. We use a nonassociative infix functor ':' to separate keys from other data.

An inorder traversal of a BST visits the nodes in increasing order, according to the ordering relation in the set of keys. For example, the



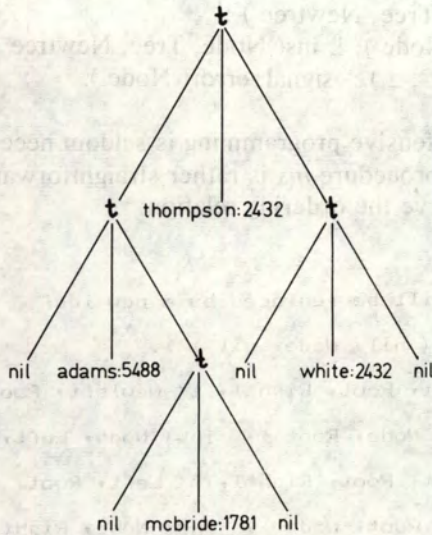


FIG. 4.3 Another BST.

following procedure can be used to write out name–phone pairs, sorted alphabetically by names:

```

write_sorted( nil ).
write_sorted( t( Left_subtree, Node_info, Right_subtree ) ) :-
    write_sorted( Left_subtree ),
    write( Node_info ), nl,
    write_sorted( Right_subtree ).

```

In this procedure, we need not test the actual ordering of nodes; this would not be the case if we wanted, say, to locate a node in a tree. Let the call

```
precedes( Node1, Node2 )
```

succeed iff Node1 comes before Node2. For our name–number pairs the procedure can be defined simply as

```
precedes( Name1 :_, Name2 :_ ) :- Name1 @< Name2.
```

It is reasonable to expect that nodes are correctly built, e.g. that each key is a name, and other information a number. A good place to check this would be a procedure for inserting a node into a tree:

```

insert( Node, Tree, Newtree ) :-
    correct( Node ), !, ins( Node, Tree, Newtree ).
insert( Node, _, _ ) :- signal_error( Node ).

```

However, such defensive programming is seldom necessary in practice.

The insertion procedure *ins* is rather straightforward. We must only take care to preserve the ordering relation:

```

% an empty tree will be replaced by a new leaf
ins( Node, nil, t( nil, Node, nil ) ).

ins( Node, t( Left, Root, Right ), t( Newleft, Root, Right ) ) :-
    precedes( Node, Root ), ins( Node, Left, Newleft ).

ins( Node, t( Left, Root, Right ), t( Left, Root, Newright ) ) :-
    precedes( Root, Node ), ins( Node, Right, Newright ).

```

The procedure fails when it tries to duplicate a key (both calls on *precedes* fail). If the keys need not be unique, we must relax one of the tests, e.g. by changing

```

    precedes(Root, Node)
into
    not precedes(Node, Root)

```

A BST can be built by successive insertions. We shall not discuss balanced trees. They present problems of their own, which can be solved by far in the same way as in classical programming languages (see e.g. Sedgewick 1983) but which can cause memory problems with some Prolog implementations. One example is an AVL-tree insertion program (van Emden 1981, Vasey 1982).

We need some thought to delete a node even from an unbalanced tree. If either of the subtrees of the deleted node is empty, the other subtree moves up and replaces the node. For example, deleting *adams* : \_ from the tree in Fig. 4.3 gives the tree in Fig. 4.4. Suppose now that both subtrees are nonempty; we shall preserve the ordering if we replace the deleted node by that with the largest key in the left subtree (or else that with the smallest key in the right subtree). For example, deleting *thompson* : \_ in Fig. 4.3 gives the tree in Fig. 4.5.



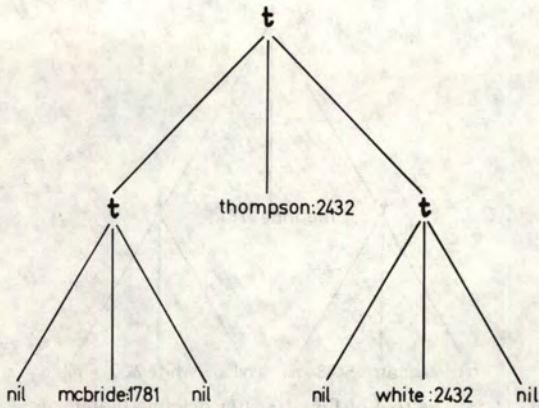


FIG. 4.4 The tree of Fig. 4.3 after deleting adams : \_ .

The following procedures implement this algorithm. The second clause is for symmetry (and for efficiency) but it is not really necessary.

```

del( Node, t( nil, Node, Right ), Right ).
del( Node, t( Left, Node, nil ), Left ).
del( Node, t( Left, Node, Right ), t( Newleft, Leftmax, Right ) ) :-
    remove_max( Left, Leftmax, Newleft ).
del( Node, t( Left, Root, Right ), t( Newleft, Root, Right ) ) :-
    precedes( Node, Root ), del( Node, Left, Newleft ).
del( Node, t( Left, Root, Right ), t( Left, Root, Newright ) ) :-
    precedes( Root, Node ), del( Node, Right, Newright ).

% find and remove the node with the largest key
remove_max( t( Left, Max, nil ), Max, Left ).
remove_max( t( Left, Root, Right ), Max, t( Left, Root, Newright ) ) :-
    remove_max( Right, Max, Newright ).
  
```

Normally we would call the procedure *del* with only the key given. We might encapsulate such calls:

```

delete( Key, Oldtree, Newtree ) :-
    del( Key : _, Oldtree, Newtree ).
  
```

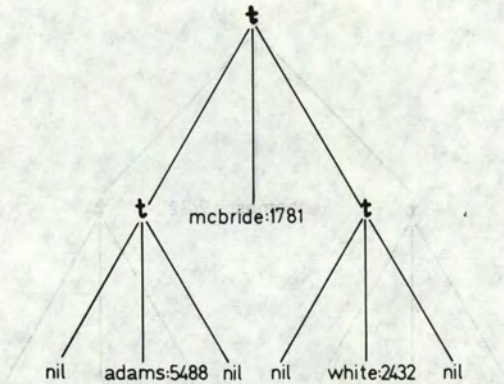


FIG. 4.5 The tree of Fig. 4.3 after deleting thompson : \_.

The last basic operation on BSTs is the search itself:

```

search( Node, t( _, Node, _ ) ) .
search( Node, t( Left, Root, _ ) ) :-
    precedes( Node, Root ), search( Node, Left ).
search( Node, t( _, Root, Right ) ) :-
    precedes( Root, Node ), search( Node, Right ).
  
```

Again, we can encapsulate typical calls—“find information associated with a given key”:

```
find( Tree, Key, Data ) :- search( Key : Data, Tree ).
```

A slightly different method of representing binary trees consists in using

```
l( Node )
```

for leaves, instead of `t(nil, Node, nil)`. However, with this representation we would have to distinguish empty trees from leaves of non-empty trees. For example, two more clauses would be necessary in the procedure for tree insertion.

As a very special case, we can consider trees of degree 1, that is, lists. Recall that a widespread convention (introduced in Chapter 1) is to denote empty lists by the atom

```
[]
```



and non-empty lists by infix terms

Head.Tail

The period is used to build trees of degree 2, which are a convenient representation of lists. It plays the same role as *t* in our BST example. In Prolog-10 a special notation has been invented as yet another application of syntactic sugar. It is very commonly used, even though its advantages over dot notation are debatable. Instead of Head.Tail we shall write<sup>2</sup>

[ Head | Tail ]

the list a.b.c.Tail will be written as

[ a, b, c | Tail ]

and the list a.b.c.[] as

[ a, b, c ]

To make sure you have mastered this notation, check that [c | [d]] is the same as [c, d].

We shall remind you of two list-manipulating procedures from Chapter 1. Membership:

```
member( Item, [ Item | Tail ] ).
```

```
member( Item, [ _ | Tail ] ) :- member( Item, Tail ).
```

And list concatenation:

```
append( [], Second, Second ).
```

```
append( [ Head | First_tail ], Second, [ Head | Third_tail ] ) :-
```

```
append( First_tail, Second, Third_tail ).
```

Here is another small example of operations on lists. Consider the following simple-minded sorting algorithm: given a list, put all its members in a BST and then apply the procedure *write\_sorted*, defined above.

```
sort( List ) :- buildtree( List, nil, Tree ), write_sorted( Tree ).
```

```
% 2nd and 3rd argument: the tree built so far, the final tree
```

```
buildtree( [], Finaltree, Finaltree ).
```

```
buildtree( [Item | Items], Currenttree, Finaltree ) :-
```

```
insert( Item, Currenttree, Nexttree ),
```

```
buildtree( Items, Nexttree, Finaltree ).
```

<sup>2</sup> Sometimes an equivalent notation is used: [Head... Tail], with ... written without blanks.

In Section 4.2.3 we shall define a more useful sorting procedure based on BSTs. It will construct the sorted permutation of a given list.

Just as in other programming languages, lists are used in Prolog primarily to represent sequences and sets. They can also be used in a standard way to represent trees of unspecified degree. For example, the tree of Fig. 4.6 might be represented by the list

```
[ a, [ b, [ e ] ], [ c ], [ d, [ f ], [ g ] ] ]
```

Lists are best utilized when items are processed sequentially from left to right, or when all processing takes place at the beginning of the list. In the latter case the list is used as a stack. The basic stack operations, *push* and *pop*, can be easily written in one procedure, e.g.

```
stack_op( Top, Rest_of_stack, [ Top | Rest_of_stack ] ).
```

with the call

```
stack_op( Newtop, Stack, Newstack )
```

serving as *push*, and the call

```
stack_op( Top, Newstack, Stack )
```

to execute *pop*. However, in practice we would rather operate on the stack implicitly, by using appropriate terms in clause heads. One example is the procedure *reduce* (see Section 7.4.3) with old and new stacks as parameters. The clause

```
reduce( [ br( r, '()' ) , t( X ) , br( l, '()' ) , id( I ) | S ],  
        [ t( tr( I, X ) ) | S ] ).
```

describes an action that consists of four pops followed by one push.

Nonsequential access to a list requires, as might be expected, time proportional to the list's length. To build a list in linear time, we can

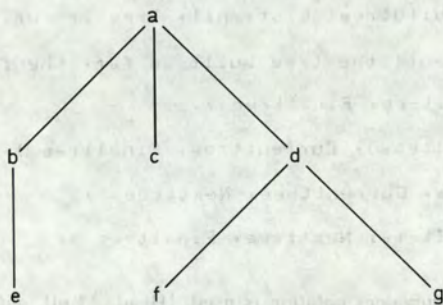


FIG. 4.6 A non-binary tree.



successively push incoming items, but the original sequence will be reversed. Alternatively, we can use *append* to preserve the original order of items, but this would square the running time. Moreover, each call on *append* entails not only a traversal of the entire list, but also creation of its copy. Strictly speaking, a series of variables is produced and instantiated to successive tails. When executing the call

```
append( [ It1, It2 ], [ It3 ], X )
```

the following instantiations take place:

```
X ← [ It1 | Third_tail' ]
Third_tail' ← [ It2 | Third_tail'' ]
Third_tail'' ← [ It3 ]
```

As a result, only the top-level structure is copied. The situation is roughly as in Fig. 4.7: the two lists share all items but the last.

We had a similar situation in the tree insertion procedure. Check that Fig. 4.8 properly illustrates the picture after inserting *turner* : 6481 into the tree of Fig. 4.3: we copy the top-level structure of the whole branch.

Copying structures upon modification is necessary because of the semantics of the operations: when we call *append*(*L1*, *L2*, *L3*) to concatenate *L1* and *L2*, we may wish to preserve an unmodified *L1*. If we want destructive modification operations, we must express this desire explicitly.

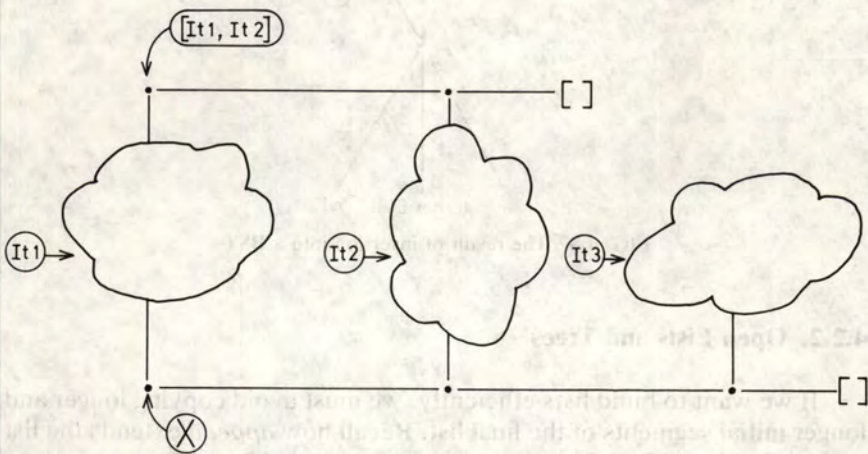


FIG. 4.7 The result of appending two lists.

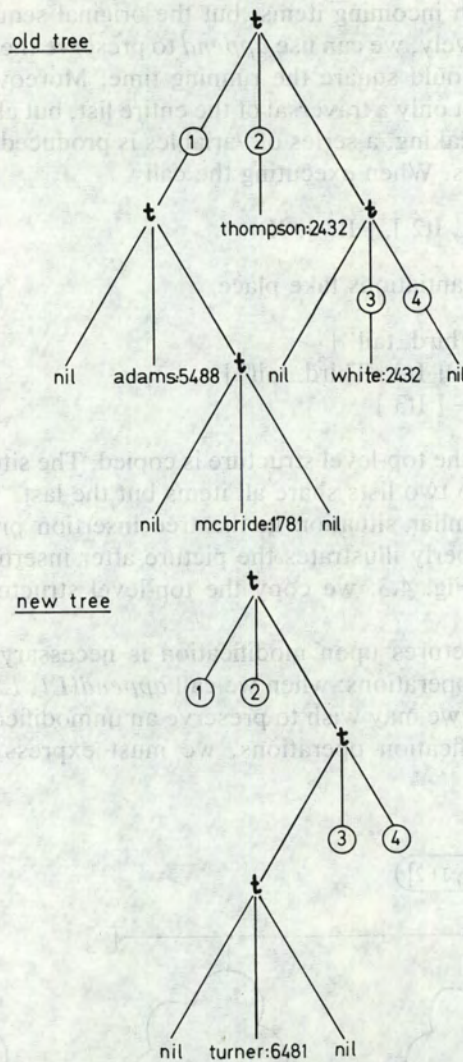


FIG. 4.8 The result of insertion into a BST.

#### 4.2.2. Open Lists and Trees

If we want to build lists efficiently, we must avoid copying longer and longer initial segments of the final list. Recall how *append* extends the list piece by piece. After the call

```
append( [ It1, It2 ], [ It3 ], X )
```



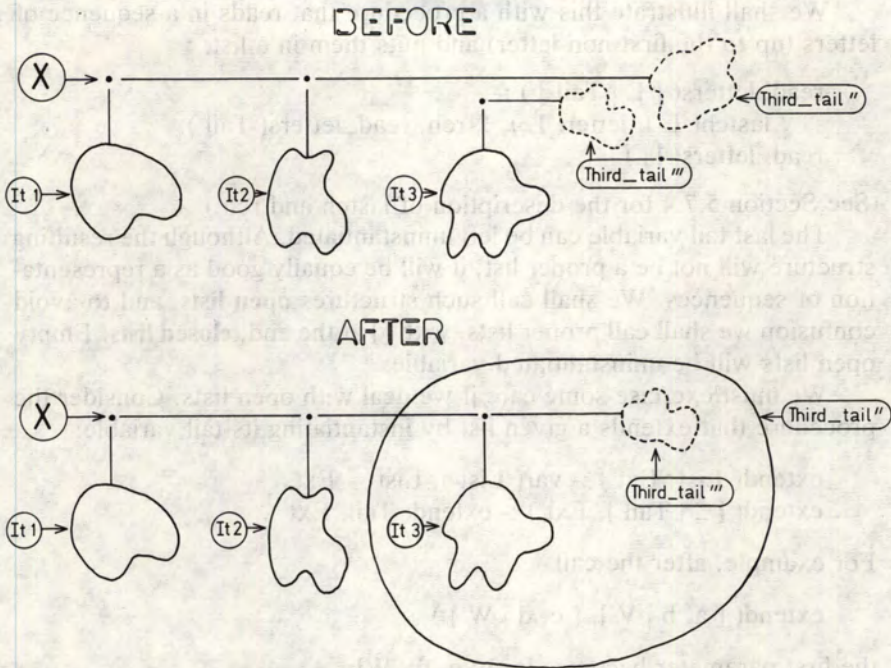


FIG. 4.9 Extending a list.

we get

$$X \leftarrow [ It1 \mid Third\_tail' ]$$

$$Third\_tail' \leftarrow [ It2 \mid Third\_tail'' ]$$

and finally bind  $Third\_tail''$ . The trick is to keep  $Third\_tail''$  ready for a subsequent instantiation:

$$Third\_tail'' \leftarrow [ It3 \mid Third\_tail''' ]$$

The situation will be roughly as in Fig. 4.9. Figuratively speaking, we shall be able to resume *append* in the next step of computation. We only need to get hold of the variable  $Third\_tail'''$ , instantiate it:

$$Third\_tail''' \leftarrow [ It4 \mid Third\_tail'''' ]$$

and so on. When we are through, we can instantiate, say,

$$Third\_tail'''' \leftarrow []$$

and come up with the final instance of  $X$ ,

$$[ It1, It2, It3, It4 ]$$

We shall illustrate this with a procedure that reads in a sequence of letters (up to the first non-letter) and puts them in a list:

```
read_letters( [ L | Tail ] ) :-
    lastch( L ), letter( L ), !, rch, read_letters( Tail ).
read_letters( [ ] ).
```

(See Section 5.7.4 for the description of `lastch` and `rch`.)

The last tail variable can be left uninstantiated. Although the resulting structure will not be a proper list, it will be equally good as a representation of sequences. We shall call such structures **open lists**, and to avoid confusion we shall call proper lists, with `[]` at the end, **closed lists**. Empty open lists will be uninstantiated variables.

We must exercise some care if we deal with open lists. Consider the procedure that extends a given list by instantiating its tail variable:

```
extend( List, Ext ) :- var( List ), List = Ext.
extend( [ _ | Tail ], Ext ) :- extend( Tail, Ext ).
```

For example, after the call

```
extend( [ a, b | V ], [ c, d | W ] )
```

the first parameter becomes `[a, b, c, d | W]`.

It is essential that the instantiation of the tail variable be delayed. Consider what would happen if we changed the first clause to (apparently equivalent)

```
extend( Ext, Ext ).
```

The result of the call

```
extend( [ X, Y, Z | End1 ], [ a, b | End2 ] )
```

(i.e. the first parameter's instantiation) would be `[a, b, Z | End1]` instead of the expected `[X, Y, Z, a, b | End2]`.

Procedure *extend* can reasonably be used only in strictly deterministic fashion. Failure after a successful computation causes dummy elements to be inserted after the first list. For example, the calls

```
extend( [ a | E1 ], [ b | E2 ] ), fail
```

instantiate `E1` as `[b | E2]`, `[_ , b | E2]`, `[_ , _ , b | E2]`, etc. Therefore a more reasonable version would be that with a cut at the end of the first clause.

The reasoning that has led us to open lists can also be applied to trees. Uninstantiated variables represent empty **open trees**. Non-empty open trees will be represented as before. For example, the following term represents the tree of Fig. 4.1 (`E1`, ..., `E6` are distinct variables):



```
t( E1, few, t( t( E2, languages, E3 ), many, E4 ),
    people, t( E5, speak, E6 ) ) )
```

Again, we shall refer to trees discussed before as **closed trees**.

We need not copy anything to insert a node into an open tree. We can go down the appropriate branch, locate a suitable empty tree, i.e. a variable, and instantiate it to a new leaf:

```
ins( Node, Empty ) :-
    var( Empty ), Empty = t( E1, Node, E2 ).

ins( Node, t( Left, Root, _ ) ) :-
    precedes( Node, Root ), ins( Node, Left ).

ins( Node, t( _, Root, Right ) ) :-
    precedes( Root, Node ), ins( Node, Right ).
```

If we rewrite the first clause as

```
ins( Node, t( E1, Node, E2 ) ).
```

a subtle change in the procedure's behaviour will ensue. The procedure will insert nothing if this Node was already present in the tree. Surprisingly it will also be identical<sup>3</sup> to the procedure *search* from the previous section, and (as might be expected) will serve almost the same purpose. The overall effect of this insertion/search procedure can be described as follows. It looks for a given Node and succeeds after finding it. However, if there is no matching node in the tree, the procedure inserts Node and then "finds" it as well.

There are some strikingly elegant applications of this. A well-known example is maintenance of symbol tables for translators written in Prolog. If the translated language is not block structured, a symbol table usually cannot contain duplicate entries, and it normally only grows, so that keeping it in an open tree will require no copying at all.

The example we are going to present is, of necessity, rather involved. Before we proceed, you might find it helpful to return to Sections 3.3 and 3.4.1, where we described a simple Algol-like language and sketched a parser and a code generator.

We intend to produce object code for a single-address target machine.

<sup>3</sup> The only difference is strictly technical: in some Prolog implementations dummy variables cannot be used to pass information, so we must insert a leaf with fresh *named* variables.

For simplicity, we assume the code will not contain external references (we shall also not attempt any optimisations).

The code generator's output should be a list of "symbolic" instructions—terms described schematically as

Opcode( Address )

Each Address is an uninstantiated variable. There should be a unique Address for every addressable symbol of the source program (variable, constant, label), and for every label created by the code generator. By way of explanation, we give a possible translation of the assignment

$$x := x + y * y + 2$$

—most opcodes have obvious meaning.

```
[ load( A1 ),
  store( A2 ),
  load( A3 ),
  mult( A3 ),
  add( A4 ),
  add( A2 ),
  store( A1 ),
  stop( _ ),
  label( A1 ), data( _ ),    % x
  label( A2 ), data( _ ),    % temporary
  label( A3 ), data( _ ),    % y
  label( A4 ), data( 2 )     % constant 2
]
```

We want the same Prolog variable for all occurrences of a source variable; for example, A1 always stands for x.

To assemble this section of code, we should determine the base address and go down the list, counting bytes (or other units of storage). Each executable instruction would be assigned a final address. The pseudoinstruction *label* would be treated differently. We would instantiate Address as the current value of the location counter (without advancing the counter); this would instantiate *all* occurrences of Address (or of variables bound to it, if one wants such fine distinctions). Assuming each instruction takes four bytes and the fragment of code starts at location 1000, we would obtain

```
1000 : load( 1032 ),
1004 : store( 1036 ),
1008 : load( 1040 ),
1012 : mult( 1040 ),
etc.
```



Conveniently enough, all we need to achieve this remarkable behaviour is the procedure *ins* (it should have rather been christened *table\_lookup*). Whenever the translator encounters a symbol, say *x*, in the source program, it allocates a fresh variable *V*, to represent the symbol in subsequent processing. It also calls *ins* to locate or place the pair

$p(x, V)$

in the symbol table. On the first occurrence of *x* the pair will actually be inserted. A subsequent "insertion" of  $p(x, U)$  only binds *V* and *U* together, i.e. finds *x*'s "symbolic address".

For this scheme to work properly, each non-terminal symbol in the grammar that implements our code generator (see Section 3.4.1) must be furnished with one additional parameter to pass the symbol table<sup>4</sup>. The whole grammar should be called with an empty table:

```
generate_code( S, O ) :-
    phrase( code( S, SymTab ), O ).
```

And here is a rule that might be used to generate code for assignments:

```
code( assign( Name, Expr ), SymTab ) →
    codeexpr( Expr, SymTab ),
    % code for this arithmetic expression,
    % the value will be left in the accumulator
    [ store( Addr ) ],
    { ins( p( Name, Addr ), SymTab ) }.
```

Symbol tables can also be implemented in open lists. For short tables lack of overhead due to key ordering tests can outweigh the loss due to worse performance. The simplest lookup procedure for open lists can be written as follows:

```
lookup( Entry, [ Entry | Tail ] ).
lookup( Entry, [ _ | Tail ] ) :- lookup( Entry, Tail ).
```

This procedure, and two other versions (a bit more sophisticated) have been used in the Prolog part of ToyProlog implementation (see Section 7.4, Appendices A.2 and A.3), and in the program described in Section 8.2.

Open lists were first used in the bootstrapped Prolog interpreter from Marseilles (Battani and Méloni 1973, Roussel 1975). The technique shown in the code generator example was presented by Colmerauer (1975, 1978). Open trees were introduced by Warren (1977b, 1980b).

<sup>4</sup> For simplicity, we omitted the symbol table while developing the parser. We can save this particular program by doing symbol table management in the back-end, but of course the more proper way is to install symbols in the table in the front-end.



### 4.2.3. Difference Lists<sup>5</sup>

If the application does not require shortening a list, open lists can be constructed with no copying whatsoever. Successive instances of the originally empty list—a variable—are longer and longer open lists (assuming, of course, that we are careful to instantiate final variables appropriately). However, each time we add an item, the list must be traversed to find the final variable. To avoid this, we can keep this variable ready for instantiation:

End = [NewItem | NewEnd ]

and make NewEnd available for further processing.

The pair consisting of a list *and* its final variable can be considered another representation of the list—a little redundant for the sake of efficiency. It is reasonable to represent the term as a single term. We shall write it as

OpenList -- ItsFinalVariable

with -- a nonassociative infix functor. For example:

[ a, b | X ] -- X

To add an item at the end of a list we use the procedure

additem (Item, List -- [Item | NewEnd], List -- NewEnd).

The call

additem( 4, [ 1, 2, 3 | X ] -- X, NewList )

instantiates, as expected,

NewList ← [ 1, 2, 3, 4 | NewEnd ] -- NewEnd

because

X ← [ 4 | NewEnd ]

Consequently, the old list becomes

[ 1, 2, 3, 4 | NewEnd ] -- [ 4 | NewEnd ]

To get a new list, we had to destroy the old one.

Fortunately, the destruction is apparent. The pair can still be regarded as a representation of the sequence 1, 2, 3. Notice that [4 |

<sup>5</sup> Difference lists (d-lists) were introduced by Clark and Tärnlund (1977).



NewEnd] is a tail of  $[1, 2, 3, 4 \mid \text{NewEnd}]$ . The sequence consists of those items we must pop off the first list to get its tail, i.e. of items by which the two lists differ—hence the name of this data structure: **difference list** (d-list for short).

Actually, a pair consisting of an open list and its tail is only a special case: a difference list is defined as a pair  $X \text{ -- } Y$  such that  $X = Y$  or  $X = [A_1, \dots, A_n \mid Y]$  for some  $n \geq 1$ . In general, no restrictions need be placed on the form of  $Y$ , although the most interesting applications of difference lists are those where  $Y$  is an open list.

Difference lists can be used to advantage whenever activity is expected at both ends of the sequence, e.g. when it is used as a queue. The procedure *additem* enqueues an item. To dequeue an item, we can use the obvious

```
remitem( Item, [ Item | List ] -- End, List -- End ).
```

but the behaviour of this procedure is unsatisfactory for empty lists. The call

```
remitem( Item, E -- E, NewList )
```

instantiates *NewList* as  $\text{List} \text{ -- } [\text{Item} \mid \text{List}]$ , i.e. as a “negative difference list”<sup>6</sup>. A procedure which fails, given an empty list, may be written as follows:

```
removeitem( Item, List -- End, NewList -- End ) :-  
    not List = End, List = [ Item | NewList ].
```

Another nice feature of difference lists is the way they can be concatenated. Suppose we have two lists:

```
[ a, b | X ] -- X    and    [ c, d, e | Y ] -- Y,
```

and we want to compute a list holding the sequence  $a, b, c, d, e$ . If we can assure that

```
X = [ c, d, e | Y ],
```

we shall have  $[a, b \mid X] = [a, b, c, d, e \mid Y]$ , and

```
[ a, b | X ] -- Y
```

will be a solution. This is readily generalized as a procedure:

```
d_conc( List1 -- Tail1, Tail1 -- Tail2, List1 -- Tail2 ).
```

<sup>6</sup> This structure can be very useful in its own rights; see Shapiro (1983b, Section 4.8).

Once again, it must be stressed that modification of such lists is destructive. For example, the second call below fails, because  $[c, d, e \mid Y]$  does not match  $[p \mid Z]$ :

```
d_conc( [ a, b | X ] -- X, [ c, d, e | Y ] -- Y, ABCDE ),
d_conc( [ a, b | X ] -- X, [ p | Z ] -- Z, ABP ).
```

We now return to the sorting algorithm based on BSTs (see Section 4.2.1). Instead of traversing the tree, built of a given list, and merely writing out the nodes, we would rather traverse it in order to construct the sorted permutation of the list:

```
tree_sort( List, SortedList ) :-
    buildtree( List, nil, Tree ),
    buildlist( Tree, SortedList ).
```

The procedure `buildlist` “flattens” the tree (see Fig. 4.10 for an example). The general outline of the algorithm is rather obvious: we flatten the subtrees (recursively) and concatenate the resulting lists together with the root in between. Difference lists can be used to avoid numerous appends. Let the results of recursive calls be denoted by

`LFlat -- LFlatE` and `RFlat -- RFlatE`

The algorithm is programmed as follows:

```
flatten( nil, X -- X ).
flatten( t( L, Root, R ), Flat ) :-
    flatten( L, LFlat -- LFlatE ),
    flatten( R, RFlat -- RFlatE ),
    d_conc( LFlat -- LFlatE, [ Root | X ] -- X, A ),
    d_conc( A, RFlat -- RFlatE, Flat ).
```

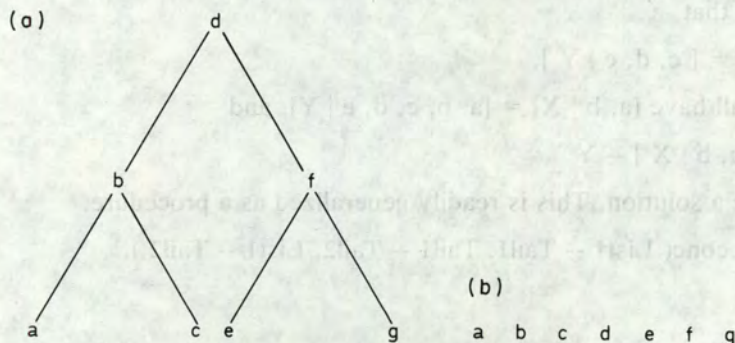


FIG. 4.10 (a) A tree. (b) The tree flattened.



This version is good for didactic purposes. Actually, we know that the following instantiations take place:

```
LFlatE ← [ Root | X ],    A ← LFlat -- X,
X ← RFlat,    Flat ← LFlat -- RFlatE
```

We can remove both calls on *d\_conc* and end up with an equivalent form of the second clause:

```
flatten( t( L, Root, R ), LFlat -- RFlatE ) :-
    flatten( L, LFlat -- [ Root | RFlat ] ),
    flatten( R, RFlat -- RFlatE ).
```

We might similarly derive a “short cut” clause for leaves. We begin with

```
flatten( t( nil, Root, nil ), LFlat -- RFlatE ) :-
    flatten( nil, LFlat -- [ Root | RFlat ] ),
    flatten( nil, RFlat -- RFlatE ).
```

then make  $LFlat = [Root | RFlat]$  and  $RFlat = RFlatE$ , and remove the recursive calls. The special case becomes:

```
flatten( t( nil, Root, nil ), [ Root | RFlatE ] -- RFlatE ).
```

(as expected!).

After the call

```
flatten( Tree, List -- ListEnd )
```

we shall have *List* instantiated as

```
[ Node1, ..., Noden | ListEnd ],
```

and all we shall need to get *SortedList* is close *List* by binding *ListEnd* to []. This is easily achieved by defining

```
buildlist( Tree, SortedList ) :-
    flatten( Tree, SortedList -- [] ).
```

(or replacing the *buildlist* call in *tree\_sort*, for that matter).

See Section 7.4.1 for a little more sophisticated application of difference lists.

#### 4.2.4. Clausal Representation of Data Structures

A Prolog procedure built of unit clauses is a natural representation of sets and sequences. Under the static interpretation of programs, such a

procedure models a relation, i.e. a set of tuples for which a certain relationship holds. For example:

```
name_phone( thompson, 2432 ).
name_phone( adams, 5488 ).
name_phone( white, 2432 ).
name_phone( mcbride, 1781 ).
```

In practice, unit clause procedures are sequences rather than sets, in that they are accessed sequentially. It is therefore possible to represent a list by a procedure, e.g.

```
list( b ).
list( k ).
list( q ).
list( y ).
```

The call

```
list( X )
```

tests membership for instantiated X, and serves as a generator for uninstantiated X. The whole list can be processed thus:

```
process_list :- list( X ), process_item( X ), fail.
process_list.
```

In general, clauses may be used to represent multidimensional matrices—we shall discuss this briefly in the next section.

The restriction to unit clauses is not essential. The clause

```
name_phone( X, 4396 ) :- office( X, room119 ).
```

will generate tuples one at a time, exactly as the other four clauses do. It is worth emphasizing that explicit and generated data are functionally indistinguishable. If five people sit in room 119, we can get up to nine name-phone pairs, without ever becoming aware of the “indirection” in one of the clauses.

Any structure expressible in terms of relations can be naturally cast in clauses. For example, a tree can be described as follows:

```
t( node1, node2, thompson : 2432, node3 ).
t( node2, nil, adams : 5488, node4 ).
t( node3, nil, white : 2432, nil ).
t( node4, nil, mcbride : 1781, nil ).
root( node1 ).
```



In particular, we can represent a list in this way:

```
l( item1, b, item4 ).
l( item2, y, nil ).
l( item3, q, item2 ).
l( item4, k, item3 ).
head( item1 ).
```

In general, every graph can be expressed as a unit-clause procedure. By way of explanation, here is a possible representation of the graph of Fig. 4.11 (see also Fig. 3.1):

```
edge( e1, e2, o ).
edge( e1, e2, letter ).
edge( e1, e3, atom ).
edge( e2, e3, x ).
edge( e2, e3, letter ).
edge( e2, e3, atom ).
```

And a representation of the graph of Fig. 4.15 (Section 4.4.3):

```
arc( a, b ).      arc( a, c ).      arc( b, c ).      arc( b, d ).
arc( b, e ).      arc( c, d ).      arc( c, e ).      arc( d, e ).
```

Clausal representation of trees, lists and the like is rather less convenient than representations described in previous sections. It cannot be passed as an actual parameter, so that its use can only be recommended when the bulk of data remains unchanged (see Section 8.1 for a non-trivial example). Since variables are local in clauses, clever techniques shown in Section 4.2.2 are hardly applicable here. To build and modify data dynamically (e.g. add a node to a tree), we must apply “extralogical” built-in procedures *assert*, *retract*, etc., to the detriment of static interpretation of programs.

There are advantages, too. First of all, in Prolog implementations which support clause indexing, *direct* access to components can be possible. **Indexing** consists in finding matching clauses by hashing rather than by linear search, so that e.g. a node in a “tree” with  $n$  nodes can be located in constant time rather than in  $\log_2 n$  steps (on the average).

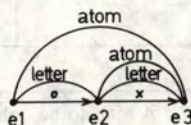


FIG. 4.11 A graph.

DEC-10 Prolog was the first to offer this possibility. If absent, it can be mimicked by means of the built-in procedure `=..` (see the next section).

Clausal representation sometimes helps reduce the problem at hand to its bare essence. A case in point is an amazingly concise solution to a map colouring problem; we quote it after Pereira and Porto (1980b). A planar map is to be coloured with at most four colours so that contiguous regions are coloured differently. First we define the contiguity relation for colours:

```
next( red, blue ).      next( red, green ).      next( red, yellow ).
next( blue, red ).      next( blue, green ).      next( blue, yellow ).
next( green, red ).      next( green, blue ).      next( green, yellow ).
next( yellow, red ).      next( yellow, blue ).      next( yellow, green ).
```

The original map of Pereira and Porto (1980b) is shown in Fig. 4.12. A region is represented by its colour—this decision makes the solution beautifully terse. To find a colouring (if any) of the map, we must only call

```
:- next( R1, R2 ), next( R1, R3 ), next( R1, R5 ), next( R1, R6 ),
   next( R2, R3 ), next( R2, R4 ), next( R2, R5 ), next( R2, R6 ),
   next( R3, R4 ), next( R3, R6 ), next( R5, R6 ),
   write( (R1, R2, R3, R4, R5, R6) ), nl.
```

Structures represented by terms are usually traversed and manipulated by recursive procedures. Clauses are traversed by backtracking, either implicit (e.g. in the call above), or explicit (e.g. in the procedure

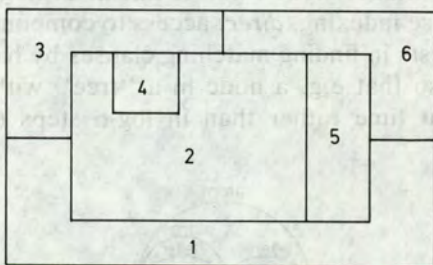


FIG. 4.12 A map to be coloured.



*process\_list*). There is a fundamental discrepancy between these two modes, because backtracking *destroys* variable instantiations which are essential to recursive operations on data structures. Consider the task of computing a list of arcs exiting vertex *b* of the graph in Fig. 4.15. Arcs are available one at a time to a routine that "backtracks through" the procedure *arc*. If we want them to survive backtracking, we must "put aside", i.e. assert, those which contain *b*:

```
put_aside :- arc( X, Y ), put_aside_if_b( X, Y ), fail.

put_aside_if_b( X, Y ) :- has( b, X, Y ), assert( with_b( X, Y ) ).

has( X, X, _ ) :- !.
has( X, _, X ).

:- put_aside.

% Now, a list can be created as follows:

collect_with_b( ThisList, Finallist ) :-
    retract( with_b( X, Y ) ), !,
    collect_with_b( [ (X, Y) : ThisList ], Finallist ).
collect_with_b( Finollist, Finallist ).

:- collect_with_b( [], TheList ), write( TheList ), nl.
```

Such operations are usually cast in terms of a general-purpose procedure that finds a set of all items for which a given condition holds. In our example, items would be  $(X, Y)$ , and the condition

$( \text{arc}( X, Y ), \text{has}( b, X, Y ) )$

The set is represented by a list, possibly with repetitions, so that it is called *bag* in the folklore. Here is our version of the procedure:

```

basof(Item, Condition, _) :-
    assert('BAG'('BAG')),           % a marker
    Condition,                       % generates an instance of Item
    assert('BAG'(Item)),             % saves it
    fail.                            % this clause eventually fails

basof(_, _, Bag) :-
    retract('BAG'(Item)), !,         % get the last Item saved
    collect(Item, [], Bag).

collect('BAG', FinalBag, FinalBag) :- !. % this was the marker
collect(Item, ThisBag, FinalBag) :-
    retract('BAG'(NextItem)), !,
    collect(NextItem, [Item : ThisBag], FinalBag).

```

The marker enables us to use the procedure *basof* within *Condition*. An example of such nested computation is the following pair of calls (*Graph* is to be a list of “bunches”—lists of arcs entering or leaving a given vertex; the condition in the first call is an alternative, in the embedded one a conjunction):

```

:- basof( X, ( arc(X, _) ; arc(_, X) ), Vertices ),
    basof( Bunch,
        ( member(U, Vertices),
          basof( (Y, Z),
              ( arc(Y, Z), has(U, Y, Z) ),
              Bunch
            )
        ),
        Graph ).

```

Repetitions in a bag may be undesired. For example, the first call above should rather find a *set* of all vertices—as it stands, *Graph* will contain numerous duplicates. The procedure *setof* would call *basof* and then filter the resulting bag. In Prolog-10 and some other implementations



both *bagof* and *setof* are built-in procedures: *setof* even returns its output sorted. An implementation of *setof* in Prolog was presented by Pereira and Porto (1981).

#### 4.2.5. Array Analogues in Prolog

There is no addressing mechanism in Prolog, no memory cells directly available to the programmer—for most applications this is simply unnecessary. Consequently, there are no arrays interpreted as contiguous, addressable areas. From a mathematical standpoint, arrays correspond to finite matrices, i.e. to mappings from finite sets of subscripts to sets of values. In theory, there are no restrictions on the form of subscripts, although integers are most commonly used.

In Prolog we can represent such mappings as procedures consisting of unit clauses, one clause for each sequence of subscripts and the corresponding value. This is but a special case of relation in the relational model of data (see Section 8.2).

Unit clauses are particularly convenient as a representation of sparse matrices, provided that clause indexing is supported by the Prolog implementation.

Another possibility is to represent a mapping as a list of  $n$ -tuples (subscripts, value), and to use list manipulation procedures. As a special case, a sequence subscripted by consecutive integers may be represented as a list of values. This approach may work for short lists, but in general it is prohibitively inefficient.

We shall now present an alternative way of storing integer-subscripted sequences, which is rather unlikely to outperform Prolog data bases (with indexing), but may be reasonable for sequences of moderate size. The method makes use of digital search trees (see e.g. Sedgewick 1983).

Branching in digital search trees is based on the values of successive digits of the key being looked for. Keys cannot be negative. The order of every node is equal to the base of the digital system, e.g. to 10 if keys are expressed in decimal. In Fig. 4.13 we show two trees, each containing 15 items numbered 0 through 14.  $A_i$  denotes the  $i$ -th item, the root is empty (i.e. contains a dummy value), and branches are labelled with digits. To find  $A_{13}$  in the binary tree, we take 1101, the binary code of 13, and go down selecting branches labelled with 1, 1, 0 and 1. To find  $A_{13}$  in the ternary tree, we use 111, the ternary code of 13.

Digital search trees are best implemented in Prolog by open trees. We shall demonstrate it in the case of ternary trees (other cases are basically

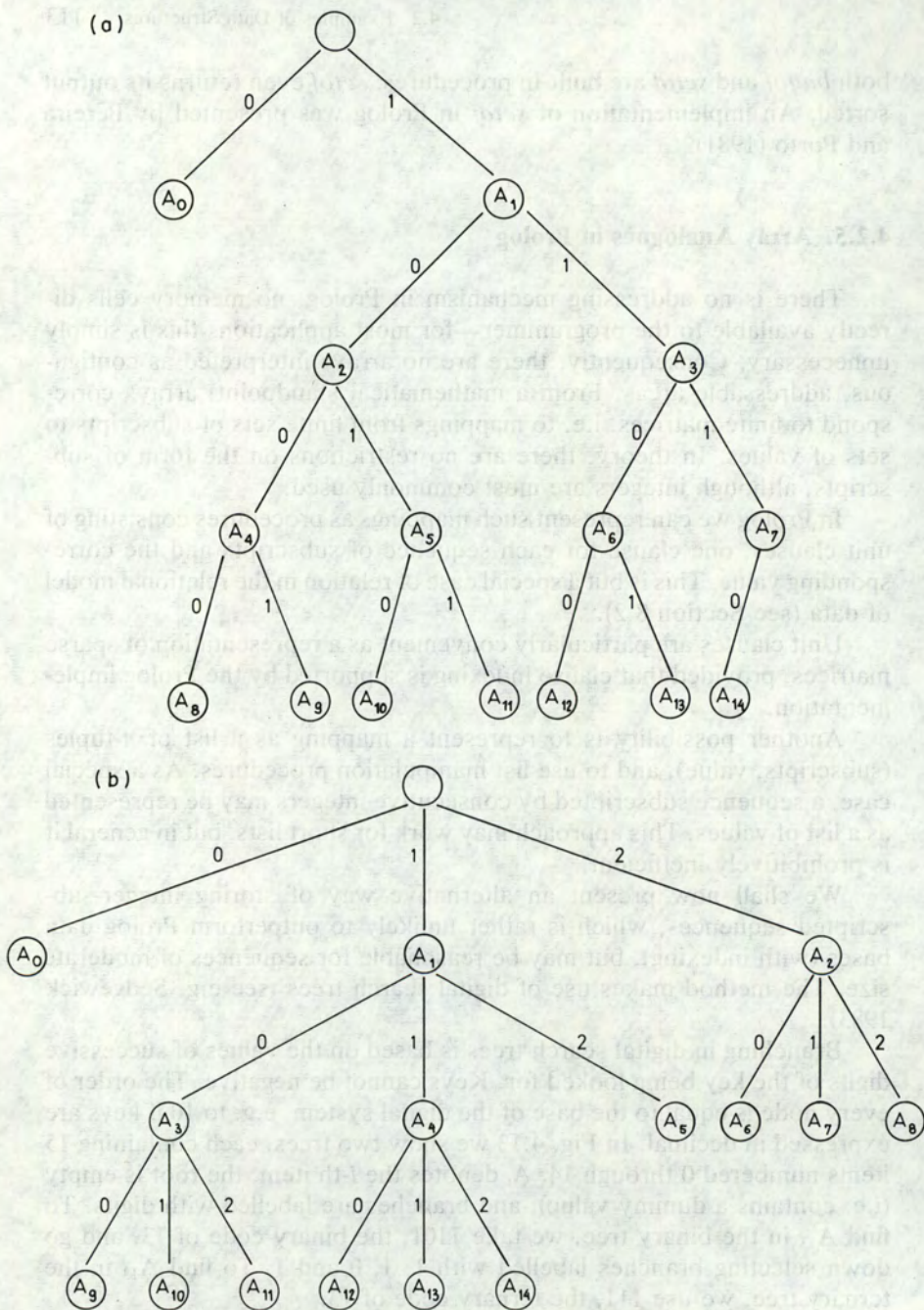


FIG. 4.13 Digital search trees: (a) Binary digital search tree. (b) Ternary digital search tree.



identical, although impractical if nodes have more than a few branches). A non-empty tree will be represented as

`t3( Value, Left, Middle, Right )`

and an empty tree as a variable. Explicit labels are unnecessary, as we may simply select Left or Middle, or Right upon encountering 0, 1 or 2, respectively.

A procedure that finds a value, given a ternary subscript and a tree, is quite straightforward. We assume that subscripts are represented as closed lists of digits:

```
find_3( [], t3( Value, _, _ , _ ), Value ).
find_3( [0 : Sub], t3( _, Left, _ , _ ), Value ) :-
    find_3( Sub, Left, Value ).
find_3( [1 : Sub], t3( _, _, Middle, _ ), Value ) :-
    find_3( Sub, Middle, Value ).
find_3( [2 : Sub], t3( _, _, _ , Right ), Value ) :-
    find_3( Sub, Right, Value ).
```

The procedure fails if the first parameter is not a correct ternary subscript, or if the second parameter is not an open ternary tree. However, it does *not* fail when a nonexistent item is referred to. We shall discuss this phenomenon presently.

Now for a procedure that replaces an item. Two tree parameters are required, and the new tree is a copy of the old one, except for the replaced item. The amount of copying is similar to that illustrated in Fig. 4.8.

```
change_3( [], NewVal, t3( _, L, M, R ),
    t3( NewVal, L, M, R ) ).
change_3( [0 : Sub], NewVal, t3( OldVal, L, M, R ),
    t3( OldVal, NewL, M, R ) ) :-
    change_3( Sub, NewVal, L, NewL ).
change_3( [1 : Sub], NewVal, t3( OldVal, L, M, R ),
    t3( OldVal, L, NewM, R ) ) :-
    change_3( Sub, NewVal, M, NewM ).
```

```

change_3( [ 2 : Sub], NewVal, t3( OldVal, L, M, R ),
          t3( OldVal, L, M, NewR ) ) :-
    change_3( Sub, NewVal, R, NewR ).

```

Both procedures behave in the same way when the subscript is too large: they create a missing part of the tree, and then “find” or “change” the newly inserted item. For example, the call

```
find_3( [ 2, 1, 0, 1 ], Tree, A64 )
```

applied to the tree of Fig. 4.13b changes the node with item A7 into the tree of Fig. 4.14, or, in term notation, into

```

t3( A7, t3( Dummy21, Empty_i,
            t3( A64, Empty_ii, Empty_iii, Empty_iv ),
            Empty_v ),
    Empty_vi, Empty_vii )

```

The same effect will be achieved by the call

```
change_3( [ 2, 1, 0, 1 ], A64, Tree, Tree )
```

The moral is that, first, no special insertion procedure is needed, and, second, the tree need not be full. It will contain only the inserted nodes together with the branches required to reach these nodes, but intermediate nodes may contain no meaningful information.

To make the story complete, here is a little procedure that converts nonnegative integers into lists of ternary digits. Note that there are two procedures here: conv\_3/2 and (auxiliary) conv\_3/3.

```

conv_3( 0, [0] ).
conv_3( N, TerN ) :- integer( N ), 0 < N, conv_3( N, [], TerN ).

conv_3( 0, AllDisits, AllDisits ) :- !.
conv_3( N, Z, AllDisits ) :-
    Digit is N mod 3, Nby3 is N / 3,
    conv_3( Nby3, [Digit : Z], AllDisits ).

```

#### 4.2.6. Access to the Structure of Terms

In Section 4.2.1 we dismissed the possibility of representing tree nodes with main functors: the term

```
few( nil, people( many( languages, nil ), speak ) )
```



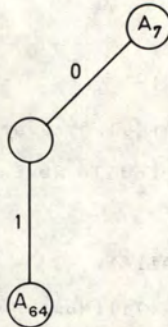


FIG. 4.14 Creation of the missing part of a tree.

would stand for the BST of Fig. 4.1. We shall now show an insertion routine for such trees. The built-in procedure `=.. (univ)` is used to circumvent the problem raised by the potential diversity of the functors.

```

insert( Node, Tree, NewTree ) :-
    Tree =.. [Root, Left, Right],
    insert( Node, Root, Left, Right, NewLeft, NewRight ),
    NewTree =.. [Root, NewLeft, NewRight].

insert( Node, nil, Node ).

insert( Node, Leaf, NewTree ) :-
    insert( Node, Leaf, nil, nil, Left, Right ),
    NewTree =.. [Leaf, Left, Right].

insert( Node, Root, L, R, NewL, R ) :-
    precedes( Node, Root ), insert( Node, L, NewL ).
insert( Node, Root, L, R, L, NewR ) :-
    precedes( Root, Node ), insert( Node, R, NewR ).
  
```

This application of *univ* is far from typical. As a more realistic example, consider the problem of translating an arithmetic expression into reverse Polish form, e.g.

$y * \text{sqrt}(\text{sqr}(x) + f(1, y))$

into

`[ y, x, sqr, 1, y, f, '+', sqrt, '*' ]`

Here is a possible solution:

```

revpol( Expr, RevExpr ) :-
    Expr =.. [Fun | Args],   revargs( Args, [], RevArgs ),
    append( RevArgs, [Fun], RevExpr ).

revargs( [], RevAll, RevAll ).
revargs( [Arg | Args], RevTillNow, RevAll ) :-
    revpol( Arg, RevArg ),
    append( RevTillNow, RevArg, RevOneMore ),
    revargs( Args, RevOneMore, RevAll ).

```

We could use difference lists to decrease the cost of multiple appendings, but the procedures would become even less readable (but try it—this would be an application of the “flatten” schema, although a little unwieldy because of the unknown number of arguments). However, a readable version would not only be much longer, but also less flexible:

```

revpol( A + B, RevExpr ) :-
    revpol( A, RevA ), revpol( B, RevB ),
    append( RevA, RevB, Aux ), append( Aux, [ '+' ], RevExpr ).
.....
revpol( sin( A ), RevExpr ) :-
    revpol( A, RevA ),
    append( RevA, [ sin ], RevExpr ).
.....
revpol( Atom, [ Atom ] ).

```

This is a closed schema: to be able to recognize a new function or operator, we must add a branch to this “case statement”.

Perhaps one of the most important applications of *univ* (and related built-in procedures) is in bootstrapped implementations of Prolog. A basic interpreter (see Chapter 6 and Section 7.3) may support Prolog (with a very rudimentary syntax) furnished with built-in procedures analogous to *call* and *univ*. Various user interfaces can then be written in this simplified Prolog (see Section 7.4), provided we can convert texts to terms.

Assume we input the text

```
foo( fie( X ), ok, X )
```



and produce its (intermediate) representation:

```
[[f, o, o], [[f, i, e], V], [[o, k]], V]
```

with uninstantiated V. (Try to write this reading program: a symbol table such as those described in Section 4.2.2 must be used to handle variable names properly.) Now we can glue the intermediate representation together:

```
glue( Inter, Inter ) :- var( Inter ), !.

glue( [FunChars : InterArgs], Term ) :-
    not alldisits( FunChars ),
    glueargs( InterArgs, Args ),
    pname( Fun, FunChars ), Term =.. [Fun : Args].

glue( [Disits], Number ) :-
    alldisits( Disits ),
    pnamei( Number, Disits ).

glueargs( [], [] ).
glueargs( [InterArg : InterArgs], [Arg : Args] ) :-
    glue( InterArg, Arg ), glueargs( InterArgs, Args ).

alldisits( [Disit : Disits] ) :-
    disit( Disit ), !, alldisits( Disits ).
alldisits( [] ).
```

The procedure *glue* should be called with the second parameter uninstantiated.

In implementations that do not support indexing (see Section 4.2.4), *univ* helps avoid linear search of matching clauses. Consider, for example, a natural language application program which maintains a dictionary whose entries can look as follows:

```
dict( program, noun( inanim ) or verb( intrans ) ).
```

```
dict( modular, adj ).
```

```
dict( an, article( indef ) ).
```

Next, assume that each word on input is filtered through this dictionary:

```
input_a_word( W, Features ) :-
    read_a_word( W ),
    ( dict( W, Features ), !; signal_unknown( W ) ).
```

Without indexing, dictionary lookup requires time proportional to the number of entries. Access to a procedure, i.e. to its first clause, usually requires approximately constant time (some form of hashing is used). We can have our dictionary in the form

```
program( noun( inanim ) or verb( intrans ) ).
modular( adj ).
an( article( indef ) ).
```

and define *dict* as

```
dict( W, Features ) :-
    Entry =.. [ W, Features ], Entry.
```

or—equivalently—as

```
dict( W, Features ) :-
    functor( Entry, W, 1 ), Entry, arg( 1, Entry, Features ).
```

A particularly simple dictionary is a table of keywords for a scanner of, say, Pascal:

const.	type.	array.	record.
function.	var.	begin.	do.

etc. To create the representation of a source program name, we can use this procedure:

```
key_or_id( Name, keyword( Name ) ) :- Name, !.
key_or_id( Name, ident( Name ) ).
```

As a final example, here is the crucial part of a definition of the procedure *phrase* which initiates processing based on a metamorphosis grammar:

```
phrase( InitialNonterminal, Terminals ) :-
    InitialNonterminal =.. [ Name | Parameters ],
    InitialCall =.. [ Name, Terminals, [] | Parameters ],
    InitialCall.
```

Note that input and output parameters are added at the beginning of the parameter list (rather than at the end, as suggested in Section 3.3).



### 4.3. SOME PROGRAMMING HINTS

We have collected here some down-to-earth suggestions which may help improve your coding technique in Prolog. Although style is largely a matter of taste, some of the things we have to say have long been present in Prolog folklore, and we feel fairly confident they are worth presenting.

#### 4.3.1. Using the Cut Procedure

Essentially, the cut commits the currently executing procedure to whatever it might have done since its activation. This is precisely what makes the cut a controversial feature: that it can only be interpreted dynamically. On the other hand, the variety of its uses and its power make it an important factor in the emergence of Prolog as a practical programming language.

In Chapters 1 and 2 we discuss the cut—in a very general manner—both as an extralogical mechanism and as a tool for improving efficiency. Here, we shall concentrate on its applications.

Despite Prolog's inherent nondeterminism, the usual computation is mostly deterministic: the majority of procedures are expected to produce a single, well-defined response to any particular set of input data. Most procedures are strictly deterministic: at most one clause of a procedure applies, regardless of the actual data.

With the procedural interpretation of Prolog in mind, clauses are commonly written as

*head* :- *tests*, *actions*.

A clause is executed for its *actions* which can be performed if and only if *head* matches the call and all *tests* succeed. This conforms to the fundamental notion of guarded commands (Dijkstra 1975). Some Prolog dialects, e.g. IC-Prolog (Clark *et al.* 1982b), even provide special syntax for "guards".

If, during a deterministic computation, *tests* have succeeded, a cut executed immediately after *tests* commits our choice of the clause. The cut saves us further—unnecessary—attempts to execute the procedure in the case of a failure later on. As an example of this fairly typical situation, consider the following:

```
% Retrieve ( fetch ) the grammatical description of a word,
% fail if there is no such word in the dictionary.
```

```

% The word may be given as a string:
find( String, Description ) :-
    isletterstring( String ),      % yes, a string
    pname( Word, String ), fetch( Word, Description ).
% or as a word, i.e. nullary functor:
find( Word, Description ) :-
    isword( Word ),               % yes, a word
    fetch( Word, Description ).
% Reject bad data:
find( Bad, _ ) :-
    not isletterstring( Bad ),
    not isword( Bad ),           % yes, bad data
    signal( Bad ), fail.

```

In this procedure, cuts may be safely placed after tests. Notice, however, that a cut inserted *earlier* changes the procedure's behaviour, and a cut after *fetch* does not work if a word is absent from the dictionary. (It would also have ruinous effects if *fetch* were a nondeterministic generator of synonyms.)

When we adhere to the "guarded command" style of programming, the built-in procedure *not* is frequently used to invert tests (but see the beginning of the next section for a brief discussion of *not*'s peculiarities!). However, we would not like to perform expensive tests twice, as in this example:

```

addunique( Item, List ) :-
    presentalonglist( Item, List ), signal_dupl( Item ).
addunique( Item, List ) :-
    not presentalonglist( Item, List ), additem( Item, List ).

```

We can replace the inverted test with a cut after the original test:

```

addunique( Item, List ) :-
    presentalonglist( Item, List ), !,
    signal_dupl( Item ).
addunique( Item, List ) :-      % not present...( Item, List )
    additem( Item, List ).

```

This procedure can be interpreted as

```

if present...( Item, List ) then signal_dupl( Item )
else additem( Item, List )

```

This is, perhaps, the most frequent application of the cut. It must be remembered, though, that this use of the cut is extralogical: a clause with



a test removed means something else, and it cannot be understood in separation from the rest of the procedure. Still, the procedure as a whole is usually sufficiently readable, if we view it as a (possibly nested)

**if ... then ... else if ... then etc.**

Sometimes cuts inside a procedure are undesirable. One example is a procedure that holds data, e.g.

```
father( jack, tom ).
father( bill, john ).
etc.
```

(with empty *tests* and *actions*). With a cut in each clause this would not only look ugly, the procedure would be of no use as a generator! Instead, we should commit the *call* on *father*, as in this procedure:

```
is_father( Person ) :- father( Person, _ ), !.
```

The cut serves as a firewall against unwanted backtracking.

Another example. Consider this group of grammar rules:

```
command( Cmd ) → stop( Cmd ).
command( Cmd ) → dump( Cmd ).
command( Cmd ) → load( Cmd ).
command( Cmd ) → create( Cmd ).
etc.
```

A “switch” such as *command* is best committed by the cut after a call, e.g.

```
phrase( command( Cmd ), Tokens ), !
```

This technique, however, has a disadvantage. The “committing” cut affects not only the call but also the calling procedure. If the call being committed happens to be the last *test* in a clause, then the cut plays two roles at once. Otherwise we should make it invisible to the surrounding clause. To achieve this, we can use this general-purpose “call-and-commit” procedure:

```
once( Call ) :- Call, !.
```

Other arguments against “cutting high” are implementation-dependent. First of all, in many implementations memory requirements are smaller when there are fewer fail-points, so it may be desirable to perform cuts as soon as possible. Some implementations also optimise storage utilisation of tail-recursive procedures (see Section 6.4). A procedure may

become tail-recursive dynamically, after having its remaining clauses cut off. For example:

```
% Recognize a sequence of letters/digits.
ld( [ Ch | Chs ] ) → [ Ch ], { letter( Ch ) }, !, ld( Chs ).
ld( [ Ch | Chs ] ) → [ Ch ], { digit( Ch ) }, !, ld( Chs ).
ld( [ ] ) → [ ].
```

(Here, the cuts may protect us against deep recursion, effectively changing it into iteration.)

Sometimes the use of cuts should be recommended for clarity. We shall present two versions of the procedure that translates the term  $(A_1, \dots, A_n)$  into the list  $[A_1, \dots, A_n]$  and the term  $A$  (other than a comma-term) into  $[A]$ . First the version with “full guards”:

```
c_list( AA, [ AA ] ) :- var( AA ).
c_list( AA, [ A | As ] ) :-
    not var( AA ), AA = ( A, AATail ), c_list( AATail, As ).
c_list( AA, [ AA ] ) :-
    not var( AA ), not AA = ( _, _ ).
```

And the version with cuts (here the order of clauses is crucial):

```
c_list( AA, [ AA ] ) :- var( AA ), !.
c_list( ( A, AATail ), [ A | As ] ) :-
    !, c_list( AATail, As ).
c_list( AA, [ AA ] ).
```

In nondeterministic procedures cuts should be used cautiously, if we do not want to inadvertently lose some solutions. In particular, procedures that compute multiple answers (such as *append*) should not contain cuts. A cut after a call on a generator makes it yield only its first satisfactory answer, as in this small example:

```
int( 0 ).
int( NextN ) :- int( N ), NextN is N + 1.
:- int( X ), satisfactory( X ), !.
```

Cuts after tests in a procedure written according to the “guarded command” style implement Dijkstra’s don’t-care nondeterminism of **if** statements: any—exactly one—of the branches with true guards is chosen (in Prolog, the first one).

Special care must be exercised when adding cuts to procedures intended to be used in more than one way (such as grammar rules intended both for analysis and synthesis).



### 4.3.2. Failure as a Programming Tool

The procedure *not*, used to invert tests, owes its power and conciseness to the *combined* effect of three extralogical mechanisms in Prolog: variable calls, the cut, and forced failure. Recall the definition:

```
not X :- X, !, fail.
not _.
```

Observe that the second clause performs no instantiations, and any instantiations in *X* must have been undone on failure. If *not* succeeds, its parameter will remain intact. Therefore, *not* will not return anything. For example, the call

```
not student( X )
```

with uninstantiated *X* will not find a nonstudent (as might have been expected). Instead, it will fail if there is at least one student, e.g.

```
student( jim ).      student( jill ).
```

Otherwise it will succeed with *X* still a variable. If we insist on finding nonstudents, we can look for them among New Yorkers:

```
newyorker( tim ).    newyorker( jim ).
newyorker( jill ).    newyorker( amy ).
```

Now the command

```
:- newyorker( X ), not student( X ),
   write( X ), nl, fail.
```

will print:

```
tim
amy
```

It must be emphasized that *not* called with a term containing variables does not implement negation properly (see Clark 1978). If the call *not student(X)* succeeds, then we shall actually prove that

$$\neg \exists x \text{ student}( x )$$

which is equivalent to

$$\forall x \neg \text{student}( x )$$

On the other hand, suppose *not* means  $\neg$ . The command

```
:- not student( X ).
```

would then be interpreted (see Chapter 2) as

$$\forall x \neg \neg \text{student}(x)$$

i.e. as  $\forall x \text{ student}(x)$ . Its negation—to be proved by **reductio ad absurdum**—is

$$\exists x \neg \text{student}(x)$$

This discrepancy was commented upon, for example, by Clark and McCabe (1980a, 1980b) and Dahl (1980). In IC-Prolog (Clark *et al.* 1982b) the problem was solved by treating *not* calls with variables as erroneous. This is to say, negation in their system is only applicable to ground predicates.

Except for *not*, forced failure is used primarily for efficiency. Many Prolog implementations have no garbage collection, but upon backtracking almost all of them very efficiently recover some storage holding control information and term instances (see Chapter 6). We can take advantage of this in a few rather unobvious but effective tricks. One of them is “double *not*”.

On the face of it, the trick is pointless: the call

```
not not C
```

succeeds if and only if *C* does. We shall trace the execution of this call to show its hidden effect. Assume first that *C* succeeds; here are successive snapshots:

```
not not C
not C, !, fail
C, !, fail, !, fail
!, fail, !, fail
% the cut will commit the internal not
fail, !, fail
% RECOVER the storage used by C,
% and backtrack in the external not
SUCCESS
```

Now, let *C* fail:

```
not not C
not C, !, fail
C, !, fail, !, fail
% backtrack in the internal not,
% succeed via the second clause
```



```
!, fail
    % the cut commits the external not
```

```
fail
FAILURE
```

Since “double *not*” does not instantiate anything, it can only be used in two situations. Either we want to perform a complicated “yes/no” test (with all interesting variables already instantiated), or we are only interested in some side-effects of *C* but we want to recover storage after its execution. For readability, we usually define two procedures:

```
check( Cond ) :- not not Cond.
side_effects( Goals ) :- not not Goals.
```

One example should suffice:

```
prettyprint( Term ) :- side_effects( doprettyprinting( Term ) ).
```

Suppose now that we need instantiations produced when executing a call, and that space still matters. To preserve the results (i.e. the appropriate terms) over backtracking, we must “put them aside”. Only stored clauses are immune to failure. The following general-purpose procedure<sup>7</sup> executes a call, and at the same time “garbage collects” the storage used by the call:

```
with_gc( Call ) :-
    once( Call ), assert( 'ASIDE'( Call ) ), fail.
with_gc( Call ) :-
    retract( 'ASIDE'( Call ) ), !. % commit retract
```

This method makes sense when *assert* requires less storage than *Call*, or when the implementation has no general garbage collector but reclaims storage left by retracted clauses.

*with\_gc* can be employed in loop optimisation, which is an important application of forced failure. Essentially, recursion is the most natural Prolog counterpart of Pascal-like iteration. Consider a program that takes large chunks of an even larger text, extracts some data from them, and puts these data into an open tree. The storage for a step is worth recovering. Let *step* assert **basta**. after having encountered the final chunk. The loop can be written as follows:

```
buildtree( _ ) :- retract( basta ), !. % remove the signal
buildtree( Tree ) :-
    with_gc( step( Tree ) ), buildtree( Tree ).
```

(Find a similar solution for closed trees.)

<sup>7</sup> This technique was advocated by R. A. Kowalski at the Logic Programming Workshop in Debrecen, Hungary, 1980.



Suppose now that steps of a loop have no common terms (which would have to be passed down the loop). This means that a step is executed only for its side-effects. For example, consider the problem of reading in a Prolog program up to the clause *end*.. Let the procedure *clause\_in* perform one step: read a clause and assert it (unless it is *end*. or incorrect). The following procedure repeatedly calls on *clause\_in*, and recovers storage after each step:

```
getprog :- clause_in( Clause ), Clause = end, !.  
getprog :- getprog.
```

This loop can be made even more concise if we use a “failure screen”. This is a procedure that always succeeds nondeterministically, i.e. leaves room for yet another success:

```
repeat.  
repeat :- repeat.
```

(it is standard in some Prolog implementation). The loop can be expressed as

```
getprog :- repeat, clause_in( C ), C = end, !.
```

After *C = end* succeeds, the cut will remove the pending choice in *repeat*, and so terminate the loop.

For this technique to work, the core of the loop must be deterministic, as otherwise a failure of *C = end* would evoke another attempt to execute an already executed step. Usually it suffices to enclose the call for a step in *once*(\_):

```
getprog :- repeat, once( clause_in( C ) ), C = end, !.
```

A special form of forced failure is caused by *tagfail*<sup>8</sup>. This built-in procedure is described in Section 5.12, together with other associated procedures. They are all primarily used for error handling, as they allow bypassing of large fragments of a computation. Here we shall present an application of *tag* and *tagfail* for exiting loops.

An extremely simplified interactive executor of Prolog commands can be programmed as follows:

```
ear :- tag( loop ).  
ear.  
loop :- repeat, read( C ), once( C ), fail.
```

<sup>8</sup> It is only available in Toy (see Section 5.12), but something similar is present or can be programmed in several other implementations of Prolog.



The execution of

```
tagfail( loop )
```

terminates the loop: *tag(loop)* fails, and the second clause of *ear* promptly succeeds. With a step defined as

```
step :- read( C ), once( C ).
```

and *loop* redefined as

```
loop :- repeat, tag( step ), fail.
```

we can also exit one step by calling

```
tagfail( step )
```

#### 4.3.3. Clauses as Global Data

The program modification procedures—*assert*, *retract* and the like—are first of all used to maintain Prolog data bases (see Section 8.2). They can also be used in automodifying procedures, those which assert or retract their own clauses; this is an extremely dubious programming trick, and is not recommended, especially since such programs tend to be rather subtly implementation-dependent.

Modification procedures are also used to store so-called global data. In Prolog implementations that do not support modularisation, the data kept in program clauses (notably unit clauses) are accessible to all procedures, i.e. global. Such data are significant in Prolog because they are not affected by backtracking—see *with\_gc* in the previous section. Also, they are sometimes more convenient to handle than information passed around via parameters. One example is a “switch”—a parameterless unit clause whose presence or absence provides a simple yes/no test. For instance, we can supply terse or wordy error messages:

```
message( Code ) :- terse, short_mes( Code ), nl, !.
```

```
message( Code ) :- long_mes( Code ), nl, !.
```

```
short_mes( sym( S ) ) :- display( '?sym ' ), display( S ).
```

```
.....
```

```
long_mes( sym( S ) ) :-
```

```
    display( 'Unexpected symbol on input: ' ),
```

```
    display( S ), nl,
```

```
    display( ' The remainder of the command will be ignored.' ).
```

A switch can be easily turned on:

```
turnon( Switch ) :- Switch, !, % already on
```

```
turnon( Switch ) :- assert( Switch ).
```

and off:

```
turnoff( Switch ) :- retract( Switch ), !.  
                                % fails if Switch was off  
turnoff( _ ).                % already off
```

We can also revert the state of a switch (on  $\rightarrow$  off, off  $\rightarrow$  on):

```
flip( Switch ) :- retract( Switch ), !.  
flip( Switch ) :- assert( Switch ).
```

Switches are really cumbersome to program without clausal data. It is not difficult to rewrite *message*:

```
message( Code, terse ) :- short_mes( Code ), nl, !.  
message( Code, wordy ) :- long_mes( Code ), nl, !.
```

but the Terseness parameter ought to be carried everywhere throughout the program; and dynamic reversal of a switch can be somewhat messy.

Our final example demonstrates how assertions can be used to memorize results of expensive computations for future use. Let the procedure *integrate* perform symbolic integration of a given formula (and fail if it cannot be done). If we are going to use this procedure frequently, we may wish to avoid recomputing integrals. To this end, we should store every integral, once computed, and always try to find a ready answer before launching actual integration. Here is a possible solution:

```
integral( Expr, IExpr ) :- stored_integral( Expr, IExpr ), !.  
integral( Expr, IExpr ) :-  
    integrate( Expr, IExpr ),  
    assert( stored_integral( Expr, IExpr ) ).
```

In fact, we have thus furnished our program with a primitive learning capacity.

#### 4.4. EXAMPLES OF PROGRAM DESIGN

In this section we look at several tiny programming problems and their solutions which result from more or less formal analysis. This is not a real exercise in derivation of programs from formal specifications (see Hogger 1979; Gregory 1980; Burstall and Darlington 1977). This is, at best, an illustration of such derivation, not very rigorous and with formulae kept as simple as possible.

These particular problems present no difficulty to experienced pro-



grammers, who can readily solve them without resorting to sophisticated techniques. Simple as they are, they help demonstrate how logic formulae, which lend justification to a program designed in a traditional way, can also be viewed as the same program ("modulo" some clean transformations). Implications of this observation for logic programming are far-reaching and largely uninvestigated; see Shapiro (1983a) for fascinating examples of Prolog programs which are but a by-product of theoretical considerations.

Some of the procedures discussed below can be bi-directional, but we intentionally neglect such possibilities. As an exercise, try to discover some of their less obvious applications.

Formulae will be written according to the conventions of Prolog-10: variable names are capitalized, functor names begin with small letters.

#### 4.4.1. List Reversal

Let  $rev(X)$  denote the reverse of list  $X$ , let  $X \text{ with } A$  denote the result of attaching  $A$  at the end of list  $X$  (e.g.,  $[p, q]$  with  $r = [p, q, r]$ ). Let  $X = Y$  mean:  $X$  matches  $Y$ .

Assuming that  $X \text{ with } A$  has already been defined, a possible definition of  $rev$  is:

$$(4.1) \quad rev([]) = []$$

$$(4.2) \quad rev([A | Tail]) = rev(Tail) \text{ with } A$$

Now, recall that in Prolog we can comfortably express relations such as "the reversal of  $X$  is  $Y$ " (which implicitly defines  $Y$  as  $rev(X)$ ) without resorting to the notion of equality. To re-express (4.2) accordingly, we begin with the introduction of a new variable to denote  $rev(Tail)$ :

$$(4.3) \quad T = rev(Tail) \Rightarrow rev([A | Tail]) = T \text{ with } A$$

This formula is equivalent to (4.2). Another new variable will denote  $T$  with  $A$ :

$$(4.4) \quad T = rev(Tail) \Rightarrow (TA = T \text{ with } A \Rightarrow rev([A | Tail]) = TA)$$

which is equivalent to

$$(4.5) \quad (T = rev(Tail) \wedge TA = T \text{ with } A) \Rightarrow rev([A | Tail]) = TA$$

We shall rewrite this implication, and the formula (4.1), using  $reverse(X, Y)$  instead of  $rev(X) = Y$ , and  $attach(X, Y, Z)$  instead of  $Z = X \text{ with } Y$ :

$$(4.6) \quad \text{reverse}(\text{Tail}, T) \wedge \text{attach}(T, A, TA) \Rightarrow \text{reverse}([A | \text{Tail}], TA)$$

$$(4.7) \quad \text{reverse}([], [])$$

These two formulae are exactly the logical interpretation of the following procedure:

```
reverse([A | Tail], TA) :-
    reverse(Tail, T), attach(T, A, TA).
reverse([], []).
```

The procedure *attach* can be derived in a similar way:

$$(4.8) \quad [] \text{ with } A = [A]$$

$$(4.9) \quad [B | \text{Tail}] \text{ with } A = [B | (\text{Tail with } A)]$$

From (4.9) we can obtain

$$(4.10) \quad TA = \text{Tail with } A \Rightarrow [B | \text{Tail}] \text{ with } A = [B | TA]$$

and this (together with (4.8)) is rewritten as

$$(4.11) \quad \text{attach}(\text{Tail}, A, TA) \Rightarrow \text{attach}([B | \text{Tail}], A, [B | TA])$$

$$(4.12) \quad \text{attach}([], A, [A])$$

These derivations are by no means unique. Here is another reasoning that starts with (4.2). We first introduce *TA* to denote  $\text{rev}([A | \text{Tail}])$ , and get

$$(4.13) \quad TA = \text{rev}([A | \text{Tail}]) \Rightarrow TA = \text{rev}(\text{Tail}) \text{ with } A$$

which is equivalent to (4.2). Now we introduce *T*:

$$(4.14) \quad (TA = \text{rev}([A | \text{Tail}]) \wedge T = \text{rev}(\text{Tail})) \Rightarrow TA = T \text{ with } A$$

This is easily translated into Prolog:

```
attach(T, A, TA) :-
    reverse([A | Tail], TA), reverse(Tail, T).
```

In short: we managed to define *attach* by *reverse*, but the definition is only useful if we can define *reverse* independently of *attach*.

Another method of reversing a list stems from its interpretation as a stack (see Section 4.2.1). If we move the items of one stack onto another, they will come up in reversed order. Let *Stack1* and *Stack2* denote the stacks before this reversal. The final content of the second stack will be

$\text{rev}(\text{Stack1}) ++ \text{Stack2}$



with  $X ++ Y$  denoting the result of appending  $Y$  to  $X$ . The following two equalities define *rev* recursively:

$$(4.15) \quad \text{rev}([ ]) ++ \text{Stack2} = \text{Stack2}$$

$$(4.16) \quad \text{rev}([A | \text{Tail}]) ++ \text{Stack2} = (\text{rev}(\text{Tail}) ++ [A]) ++ \text{Stack2}$$

Now,  $++$  is associative, and  $[A] ++ \text{Stack2} = [A | \text{Stack2}]$ , so that we can transform (4.16) into

$$(4.17) \quad \text{rev}([A | \text{Tail}]) ++ \text{Stack2} = \text{rev}(\text{Tail}) ++ [A | \text{Stack2}]$$

Next, we introduce a new variable *Final*:

$$(4.18) \quad \begin{aligned} \text{Final} &= \text{rev}(\text{Tail}) ++ [A | \text{Stack2}] \Rightarrow \\ \text{Final} &= \text{rev}([A | \text{Tail}]) ++ \text{Stack2} \end{aligned}$$

Let *reverse2*( $X, Y, Z$ ) denote the formula  $Z = \text{rev}(X) ++ Y$ . From (4.15) and (4.18) we get

$$(4.19) \quad \text{reverse2}([ ], \text{Stack2}, \text{Stack2})$$

$$(4.20) \quad \begin{aligned} \text{reverse2}(\text{Tail}, [A | \text{Stack2}], \text{Final}) \Rightarrow \\ \text{reverse2}([A | \text{Tail}], \text{Stack2}, \text{Final}) \end{aligned}$$

(or, accordingly, the same in Prolog). For  $Y = [ ]$ , *reverse2*( $X, Y, Z$ ) reads  $Z = \text{rev}(X)$ , so to get the reversal of  $L$  we must call

$$\text{reverse2}(L, [ ], \text{LReversed})$$

—indeed, *Stack2* must be initially empty.

Notice that in going from (4.17) to (4.18) another direction of the implication could have been chosen. This choice would lead to the Prolog clause

$$\begin{aligned} \text{reverse2a}(\text{Tail}, [A | \text{Stack2}], \text{Final}) :- \\ \text{reverse2a}([A | \text{Tail}], \text{Stack2}, \text{Final}). \end{aligned}$$

which defined the shorter list in terms of the longer. Even though it is logically correct, operationally it is unrealistic: neither this nor (4.19) would match the initial call with non-empty  $L$ .

We shall conclude this section with an even less formal derivation of difference-list reversal. Let the list to be reversed be  $L \text{ -- } Z$ , where  $L = [A_1, \dots, A_n | Z]$ . We can write

$$\text{rev}(L \text{ -- } Z) = [A_n | X] \text{ -- } Y$$

with  $X \text{ -- } Y = \text{rev}([A_1, \dots, A_{n-1} | W] \text{ -- } W)$ . Since  $W$  is an arbitrary term, we can assume  $W = [A_n | Z]$ , so that

$$X \text{ -- } Y = \text{rev}(L \text{ -- } [A_n | Z])$$

We now express the longer list by the shorter (see *reverse2a* above!):

$$\begin{aligned} \text{rev}(L \text{ -- } [A_n | Z]) &= X \text{ -- } Y \Rightarrow \\ \text{rev}(L \text{ -- } Z) &= [A_n | X] \text{ -- } Y \end{aligned}$$

and rewrite it in Prolog, with *reverse\_d*(X, Y) instead of *rev*(X) = Y:

```
reverse_d( L -- Z, [ An | X ] -- Y ) :-
    reverse_d( L -- [ An | Z ], X -- Y ).
```

The base clause,

```
reverse_d( Z -- Z, Y -- Y ).
```

must come (i.e. be tried) first, because otherwise each call with a variable second parameter will fall into infinite recursion (you may wish to check this more thoroughly). This is where the peculiarities of Prolog come into play, and obscure the so-far clean derivation. It is even worse: we have missed one weakness of almost all Prolog implementations: the absence of so-called occur check during unification (see Section 1.2.3). Therefore, the base clause matches calls with a non-empty list as the first parameter, if only the list ends with a variable. For example, the call

```
reverse_d( [ a, b | Z ] -- Z, Rev )
```

instantiates  $Z \leftarrow [a, b | Z]$  and  $\text{Rev} \leftarrow Y \text{ -- } Y$ , contrary to our expectations. One possible remedy is to instantiate the final variable as  $[]$  before going on, but to this end both clauses of *reverse\_d* must be duplicated. The complete procedure follows.

```
reverse_d( [], Y -- Y ).
reverse_d( L -- [], [An : X] -- Y ) :-
    reverse_d( L -- [An], X -- Y ).
reverse_d( Z -- Z, Y -- Y ).
reverse_d( L -- Z, [An : X] -- Y ) :-
    reverse_d( L -- [An : Z], X -- Y ).
```

Check that even this improved version loops for “negative” lists such as  $[b] \text{ -- } [a, b]$ . As you see, difference lists are useful but can be rather tricky.

#### 4.4.2. Sorting

We shall derive three procedures to sort a closed list of integers in ascending order. The first two implement insertion sort and a very simple



transposition sort, a variation of "bubble sort". Both have running time proportional to the square of list length (but both seem passable because few Prolog applications require fast sorting procedures). The third procedure is the simplest quicksort, which takes less time but uses more space (the same justification applies).

Let  $X$  into  $Y$  denote the list that results from inserting the integer  $X$  into the ordered list  $Y$ . For example,

5 into [ 4, 7, 10 ] = [ 4, 5, 7, 10 ].

A possible definition of insertion consists of three formulae:

(4.21)  $A$  into [ ] = [  $A$  ]

(4.22)  $A > B \Rightarrow A$  into [  $B$  | Tail ] = [  $B$  | (  $A$  into Tail ) ]

(4.23)  $A \leq B \Rightarrow A$  into [  $B$  | Tail ] = [  $A, B$  | Tail ]

The formula (4.22) can be rewritten as

(4.24)  $A > B \wedge AT = A$  into Tail  $\Rightarrow A$  into [  $B$  | Tail ] = [  $B$  | AT ]

and then we can use  $\text{insert}(X, Y, Z)$  instead of  $X$  into  $Y = Z$  to get the following procedure:

$\text{insert}(A, [], [A]).$

$\text{insert}(A, [B | Tail], [B | AT]) :- A > B, \text{insert}(A, Tail, AT).$

$\text{insert}(A, [B | Tail], [A, B | Tail]) :- A \leq B.$

Now, let  $\text{sorted}(X)$  denote the sorted permutation of the list  $X$ . We can define  $\text{sorted}$  in the following way:

(4.25)  $\text{sorted}([]) = []$

(4.26)  $\text{sorted}([A | Tail]) = A$  into  $\text{sorted}(Tail)$

The latter formula can be replaced by

(4.27)  $ST = \text{sorted}(Tail) \wedge AST = A$  into  $ST \Rightarrow$

$\text{sorted}([A | Tail]) = AST$

To express it in Prolog, we shall rewrite  $\text{sorted}(X) = Y$  as  $\text{ins\_sort}(X, Y)$ , and get these two clauses:

$\text{ins\_sort}([], []).$

$\text{ins\_sort}([A | Tail], AST) :-$

$\text{ins\_sort}(Tail, ST), \text{insert}(A, ST, AST).$

The order of calls in the second clause is not accidental: the tests in  $\text{insert}$  require fully instantiated parameters, so the procedure would not work if  $\text{insert}$  came first!

Transposition sorting results from the observation that a sequence is unordered iff it contains an unordered pair of contiguous items (e.g.  $A, B$

such that  $A > B$ , if we consider the ascending order). Each step of a sorting algorithm should increase the “orderedness” of the sequence, e.g. by swapping  $A$  and  $B$ .

The following formula characterizes this sorting method:

$$(4.28) \quad L = X ++ [A, B | Y] \wedge A > B \wedge LT = X ++ [B, A | Y] \\ \Rightarrow \text{sorted}(L) = \text{sorted}(LT)$$

where  $X ++ Y$  denotes  $Y$  appended to  $X$ . We should describe explicitly the “less ordered” sequence by the “more ordered”, e.g. thus:

$$(4.29) \quad L = X ++ [A, B | Y] \wedge A > B \wedge LT = X ++ [B, A | Y] \\ \wedge SL = \text{sorted}(LT) \Rightarrow SL = \text{sorted}(L)$$

Using  $\text{trans\_sort}(X, Y)$  for  $\text{sorted}(X) = Y$ , and  $\text{append}(X, Y, Z)$  for  $X ++ Y = Z$ , we can rewrite (4.29) into Prolog:

```
trans_sort( L, SL ) :-
    append( X, [ A, B | Y ], L ), A > B,
    append( X, [ B, A | Y ], LT ), trans_sort( LT, SL ).
```

Suppose now that for no  $X, A, B, Y$  we have  $L = X ++ [A, B | Y] \wedge A > B$ , i.e. that  $L$  is either ordered or too short (and also ordered!). More formally:

$$(4.30) \quad \neg ( L = X ++ [A, B | Y] \wedge A > B ) \Rightarrow L = \text{sorted}(L)$$

When we rewrite this in Prolog, we shall drop the premise and place the resulting clause *after* the recursive one. The first two calls in that clause can be regarded as tests: does  $L$  contain a two-item subsequence, and is this subsequence unordered? The clause fails if this is not the case, and the premise of (4.30) becomes trivially true. We are left with the clause

```
trans_sort( L, L ).
```

which is exactly the required base clause: we proceed from “less ordered” sequences, so that eventually we must get an ordered permutation.

The arrangement of calls in the first clause is crucial. To begin with, we repeatedly isolate any two contiguous items (this fails if the list is too short), and we look at their ordering. The first improperly ordered pair terminates this process, and we recursively sort the “improved” sequence. The procedure is attributed to van Emden (Coelho *et al.* 1980).



The last sorting algorithm we are going to program in Prolog is the well-known quicksort (Hoare 1962). For a given sequence  $L$  and its element  $A$ , let  $\text{small}(L, A)$  denote the subsequence consisting of all items smaller than  $A$ , and  $\text{large}(L, A)$  those larger than  $A$ . Items equal to  $A$  will fall, say, into  $\text{small}(L, A)$ . The following formulae describe two possible situations:

$$(4.31) \quad \text{sorted}([A | L]) = \text{sorted}(\text{small}(L, A)) ++ A ++ \text{sorted}(\text{large}(L, A))$$

$$(4.32) \quad \text{sorted}([]) = []$$

The usual transformations of (4.31) give, for example,

$$(4.33) \quad \begin{aligned} \text{small}(L, A) = LAs \wedge \text{large}(L, A) = LAI \wedge \\ \text{sorted}(LAs) = SLAs \wedge \text{sorted}(LAI) = SLAI \Rightarrow \\ \text{sorted}([A | L]) = SLAs ++ [A] ++ SLAI \end{aligned}$$

When implementing quicksort, a standard practice is to compute  $\text{small}(L, A)$  and  $\text{large}(L, A)$  simultaneously, i.e. to introduce

$\text{partition}(L, A, LAs, LAI)$

instead of the first two equalities in (4.33). Here is the Prolog code for *partition* (it can be derived in a straightforward way):

```
partition([X | Tail], A, [X | Small], Large) :-
    X <= A, partition(Tail, A, Small, Large).
partition([X | Tail], A, Small, [X | Large]) :-
    X > A, partition(Tail, A, Small, Large).
partition([], _, [], []).
```

The formula (4.33) should be transformed in the usual way:

$$(4.34) \quad \begin{aligned} \text{partition}(L, A, LAs, LAI) \wedge \\ \text{sorted}(LAs) = SLAs \wedge \text{sorted}(LAI) = SLAI \wedge \\ SLAs ++ [A | SLAI] = \text{Sorted} \Rightarrow \\ \text{sorted}([A | L]) = \text{Sorted} \end{aligned}$$

This is directly expressible in Prolog, with  $\text{quick\_sort}(X, Y)$  denoting the equality  $\text{sorted}(X) = Y$ :

```
quick_sort([A | L], Sorted) :-
    partition(L, A, LAs, LAI),
    quick_sort(LAs, SLAs), quick_sort(LAI, SLAI),
    append(SLAs, [A | SLAI], Sorted).
quick_sort([], []).
```

In the worst case, the cost of appending sorted fragments is proportional to  $n^2$  for a list of length  $n$ . We can avoid appending altogether exactly as we did in *reverse2* in the previous section. We take the empty stack, sort  $\text{large}(L, A)$  and push  $\text{sorted}(\text{large}(L, A))$  onto the stack. Next, we stack  $A$ , and finally  $\text{sorted}(\text{small}(L, A))$ .

The formulae corresponding to (4.31), (4.32)—and similar to (4.15), (4.16)—are as follows:

$$(4.35) \quad \text{sorted}([A \mid L]) ++ \text{Stack2} = \\ \text{sorted}(\text{small}(L, A)) ++ [A] ++ \\ \text{sorted}(\text{large}(L, A)) ++ \text{Stack2}$$

$$(4.36) \quad \text{sorted}([]) ++ \text{Stack2} = \text{Stack2}$$

We can now repeat the same reasoning and replace (4.35) with

$$(4.37) \quad \text{partition}(L, A, \text{LAs}, \text{LA1}) \wedge \\ \text{sorted}(\text{LAs}) ++ [A] ++ \text{sorted}(\text{LA1}) ++ \text{Stack2} = \text{Sorted} \\ \Rightarrow \text{sorted}([A \mid L]) ++ \text{Stack2} = \text{Sorted}$$

The lefthand side equality in (4.37) must be rewritten as

$$(4.38) \quad \text{sorted}(\text{LA1}) ++ \text{Stack2} = \text{LargeStacked} \wedge \\ \text{sorted}(\text{LAs}) ++ [A] ++ \text{LargeStacked} = \text{Sorted}$$

We introduce  $q\_sort(X, Y, Z)$  for the equality  $\text{sorted}(X) ++ Y = Z$ , and get the following procedure:

```
q_sort([A | L], Stack2, Sorted) :-
    partition(L, A, LAs, LA1),
    q_sort(LA1, Stack2, LargeStacked),
    q_sort(LAs, [A | LargeStacked], Sorted).
q_sort([], Stack2, Stack2).
```

And wrap it in

```
quick_sort_2(List, Sorted) :- q_sort(List, [], Sorted).
```

This version of quicksort is also attributed to van Emden (Coelho *et al.* 1980).

Notice that the recursive calls on  $q\_sort$  can be interchanged. A partly uninstantiated stack will be appended to  $\text{sorted}(\text{small}(L, A))$ ; the other call will then fully instantiate the stack. Actually, the pairs  $\text{Sorted}$ ,  $[A \mid \text{LargeStacked}]$  and  $\text{LargeStacked}$ ,  $\text{Stack2}$  can be interpreted as difference lists. Try to derive more formally a version of quicksort with difference lists.



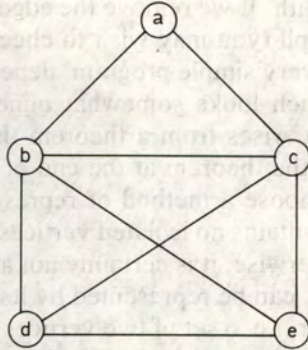


FIG. 4.15 A graph.

#### 4.4.3. Euler Paths<sup>9</sup>

We shall try to solve in Prolog the problem of finding Euler paths in an undirected graph. For the sake of completeness, here are the basic definitions. An **undirected graph** is the pair  $(V, \mathcal{E})$ , with  $V$  a finite set of **vertices** and  $\mathcal{E}$  a set of edges. A vertex is labelled with a unique name. An **edge** is an unordered pair of different vertices, usually interpreted as a connection between them. A graph is often modelled by a drawing with a point for each vertex and a line (connecting the two vertices) for each edge. Figure 4.15 shows a graph which consists of five vertices and eight edges. A **path** from vertex  $X$  to vertex  $Y$  is a sequence of edges such that contiguous edges share a vertex,  $X$  belongs to the first edge,  $Y$  to the last. For example,

$(a, b), (b, c), (c, d), (d, e)$

is a path from  $a$  to  $e$  in the graph of Fig. 4.15. The same path can be unambiguously represented as a sequence of vertices:

$a\ b\ c\ d\ e$

An **Euler path** is a path passing through all vertices, in which every edge occurs exactly once. An **Euler graph** is a graph that contains an Euler path. For our graph,

$d\ b\ a\ c\ b\ e\ c\ d\ e$

<sup>9</sup> The problem (described as "drawing a picture") was solved in Prolog by Szeredi (1977).

is an example of Euler path. If we remove the edge **bc**, the resulting graph will not be an Euler graph (you may wish to check this).

We shall develop a very simple program, depending only on the most intuitive properties, which looks somewhat blindly for Euler paths. A more efficient algorithm arises from a theorem that characterizes Euler graphs. We shall quote the theorem at the end of this section.

At first, we must choose a method of representing graphs. We can assume that the graph contains no isolated vertices (vertices which do not belong to any edge); otherwise, it is certainly not an Euler graph. A graph without isolated vertices can be represented by its set of edges alone. An edge is an unordered pair, i.e. a set of two vertices. Since we have no sets in Prolog (as in most programming languages), we shall represent edges with ordered pairs:

$$V1 \leftrightarrow V2$$

( $\leftrightarrow$  is a non-associative infix functor), and we shall try to make the program account for the commutativity of pairs.

Euler graphs have the following two properties:

1. A graph with one edge is an Euler graph.
2. Suppose we take out an edge, and what remains is a Euler graph with an Euler path starting with one of this edge's vertices; then the whole graph is an Euler graph (and we happened to have removed a terminal edge of an Euler path).

Let paths be represented with lists of vertex names, and let  $path(E,P)$  mean "E is the set of edges of an Euler graph, and P is an Euler path in this graph". The first property above can be rewritten as two formulae:

$$(4.39) \quad path(\{ V1 \leftrightarrow V2 \}, [ V1, V2 ])$$

$$(4.40) \quad path(\{ V2 \leftrightarrow V1 \}, [ V1, V2 ])$$

In other words, both arrangements of vertices are equally satisfactory.

Here is how the second property can be formalized ( $\setminus$  denotes set subtraction):

$$(4.41) \quad path(E \setminus \{ V1 \leftrightarrow V2 \}, [ V2 | RestofPath ]) \Rightarrow path(E, [ V1, V2 | RestofPath ])$$

$$(4.42) \quad path(E \setminus \{ V2 \leftrightarrow V1 \}, [ V2 | RestofPath ]) \Rightarrow path(E, [ V1, V2 | RestofPath ])$$

Before we rewrite (4.39–4.42) into Prolog, we must finally decide how to represent sets. We can use any structure capable of holding uniform data; to keep things simple we shall use lists. (Another possibility would be to represent each edge as a separate clause, but then we would have no



easy way of passing a set of edges as a parameter to a path-finding procedure.)

The formulae (4.41) and (4.42) must be transformed to get rid of the complicated expression inside *path*; for example, (4.41) becomes

$$(4.43) \quad E1 = E \setminus \{ V1 \leftrightarrow V2 \} \wedge \text{path}(E1, [V2 \mid \text{RestofPath}]) \Rightarrow \text{path}(E, [V1, V2 \mid \text{RestofPath}])$$

With the set  $\{V1 \leftrightarrow V2\}$  represented as the list  $[V1 \leftrightarrow V2]$ , and with *takeout*(*X*, *Y*, *Z*) denoting the equality  $X \setminus \{Y\} = Z$ , we can write down the procedure *path*:

```
path( [ V1 <-> V2 ], [ V1, V2 ] ).
path( [ V2 <-> V1 ], [ V1, V2 ] ).
path( E, [ V1, V2 | RestofPath ] ) :-
    takeout( E, V1 <-> V2, E1 ), path( E1, [ V2 | RestofPath ] ).
path( E, [ V1, V2 | RestofPath ] ) :-
    takeout( E, V2 <-> V1, E1 ), path( E1, [ V2 | RestofPath ] ).
```

Notice that details of set representation are transparent to the recursive clauses.

A list version of *takeout* can be defined in a straightforward manner, so we shall skip a detailed derivation:

```
takeout( [ V1 <-> V2 | E1 ], V1 <-> V2, E1 ).
takeout( [ Edge | Edges ], TheEdge, [ Edge | Remainder ] ) :-
    takeout( Edges, TheEdge, Remainder ).
```

This program is crying out for optimisation: in the worst cases we can traverse the list *E* *twice* before locating the edge to be taken out. One solution, actually presented in Szeredi (1977), is to make *takeout*, rather than *path*, sensitive to the order of vertices. This can be easily achieved by adding another base clause to *takeout*:

```
takeout( [ V2 <-> V1 | E1 ], V1 <-> V2, E1 ).
```

and deleting any one of the two recursive clauses of *path*.

The procedure *path* can be used non-deterministically, to produce all Euler paths in a given graph, or with a cut, to check whether the graph is an Euler graph (and find an instance of Euler path). It can also be used the other way round: given a path it computes a list that represents the Euler graph with this path (or all such lists, but this would be overzealous).

We shall need a few more definitions to formulate Euler's fundamental theorem on Euler graphs. A graph is **connected** if for each two vertices *V1*, *V2* there is a path from *V1* to *V2*. For example, the graph of Fig. 4.15 is connected. The **degree** of a vertex is the number of edges which contain

the vertex. For example, **b** in our graph is a vertex of degree 4, and **e** of degree 3.

The theorem states that a graph is an Euler graph if and only if it is connected and contains either no vertices of an odd degree, or exactly two such vertices. In the latter case, the two odd-degree vertices are terminal vertices of each Euler path. In the former case, each Euler path is a cycle, i.e. a path that returns to the starting point. In our example, **d** and **e** are the only vertices of odd degree.

If the graph is known to be an Euler graph, an Euler path can be found in time proportional to the number of edges. Once removed, the edge can be attached to the path for good. You may find it amusing to modify the above program in this direction.



---

## 5 SUMMARY OF SYNTAX AND BUILT-IN PROCEDURES

---

This chapter describes Prolog as defined by Toy, the implementation presented in Chapter 7. The supported dialect is very similar to Prolog-10 (Pereira *et al.* 1978, Bowen 1981, Clocksin and Mellish 1981); some, but not all, differences are noted. Other "standard" versions will be similar: use the appropriate reference manuals.

The user communicates with Toy through an interactive interface (see Sections 1.2.2 and 7.4.2).

### 5.1. PROLOG SYNTAX

A program can be regarded (roughly) as a sequence of clauses. Definitions and grammar rules in the sequence are grouped in procedures. There are quite a few principles that govern "consulting"/"reconsulting", and dynamically asserting/retracting clauses (with the *redefinition* switch on or off). Therefore a formal definition of procedures would be unnecessarily involved: it should account for the fact that procedures change in time.

The notation used is extended BNF. Non-terminal symbols will be boldfaced, and some of them subscripted (this is the first extension). A BNF rule takes the general form

$$\text{lhsnonterm} ::= \text{rhs}_1 \mid \dots \mid \text{rhs}_n$$

(**lhsnonterm** is either **rhs**<sub>1</sub> or ... or **rhs**<sub>n</sub>).

Each **rhs** is a sequence of non-terminals and terminals. The second extension: zero or more occurrences of a sequence *s* are denoted as {*s*}. To avoid confusion, terminal symbols | {} will be boldface. Throughout the description, we assume standard operator declarations are in force.

Many special forms, such as integer expressions to be evaluated by *is*,

$\mathbf{:=}$ , etc., or lists of single characters, will not be described. See Section 5.2 and on for applications of these forms in built-in procedures.

Some comments in plain English will be interspersed in the BNF description. See also the notes at the end of this section.

**clause**  $::=$  **definition** | **grammarrule** | **directive**

**definition**  $::=$  **nonunitclause** | **unitclause**

**nonunitclause**  $::=$  **head** :- **body**

**unitclause**  $::=$  **head**

COMMENT the main functor of **head** is not a binary :-

**head**  $::=$  **nonvarint**

**body**  $::=$  **bodyalt** { ; **bodyalt** }

**bodyalt**  $::=$  **call** { , **call** }

**call**  $::=$  **nonvarint** | **variable** | ( **body** )

**nonvarint**  $::=$  **term**

COMMENT not a variable or an integer (a formal definition would be straightforward but cumbersome)

**grammarrule**  $::=$  **lhside**  $\rightarrow$  **rhside**

COMMENT the arrow is written as -->

**lhside**  $::=$  **nonterminal context** | **nonterminal**

**nonterminal**  $::=$  **nonvarint**

**context**  $::=$  **terminals**

**rhside**  $::=$  **alternatives**

**alternatives**  $::=$  **alternative** { ; **alternative** }

**alternative**  $::=$  **ruleitem** { , **ruleitem** }

**ruleitem**  $::=$  **nonterminal** | **terminals** |  
condition | ! | ( **alternatives** )

**terminals**  $::=$  **list** | **string**

COMMENT only closed lists are allowed

**condition**  $::=$  **curlyterm**

**directive**  $::=$  **command** | **query**

**command**  $::=$  :- **body**

**query**  $::=$  **body**

COMMENT **body**'s main functor is not a unary :-

**term**  $::=$  **term**<sub>1200</sub>

**term**<sub>N</sub>  $::=$  **op**<sub>fx,N</sub> **term**<sub>N-1</sub> | **op**<sub>fy,N</sub> **term**<sub>N</sub> |  
**term**<sub>N-1</sub> **op**<sub>xf,N</sub> | **term**<sub>N</sub> **op**<sub>yf,N</sub> |  
**term**<sub>N-1</sub> **op**<sub>xfx,N</sub> **term**<sub>N-1</sub> |  
**term**<sub>N-1</sub> **op**<sub>xfy,N</sub> **term**<sub>N</sub> |  
**term**<sub>N</sub> **op**<sub>yfx,N</sub> **term**<sub>N-1</sub> | **term**<sub>N-1</sub>



COMMENT 1  $= < N \leq 1200$ ;  $\mathbf{op}_{\text{Type}, N}$  is an operator of type Type and priority N;  $\mathbf{term}_N$  can be called “term with priority N”

```
term0 ::= variable | integer | string |
         list | noop |
         noop( term{ , term } ) |
         ( term ) | curlyterm
```

$$\text{curlyterm} ::= \{ \text{term} \}$$

**noop ::= functor**

$$\mathbf{op}_{T,N} ::= \mathbf{functor}$$

COMMENT T is one of fx, fy, xf, yf, xfx, xfy, yfx, N is in the range 1..1200; see also note 1

$$\text{list} ::= [] \mid [ \text{term}_{999} \{ , \text{term}_{999} \} ] \mid [ \text{term}_{999} \{ , \text{term}_{999} \} \mid \text{term} ]$$

**COMMENT** terms with priority 999 can be safely  
conjoined by commas which are infix  
functors with priority 1000

$$\text{functor} ::= \text{word} \mid \text{qname} \mid \text{symbol} \mid \text{solochar}$$
$$\text{word} ::= \text{wordstart} \{ \text{alphanum} \}$$

**wordstart ::= smaller**

$$\text{alphanum} ::= \text{smallletter} \mid \text{bigletter} \mid \text{digit} \mid \_$$
$$\mathbf{qname} ::= \{ \mathbf{qitem} \}$$
$$\text{qitem} ::= ' \mid \text{nonquote}$$

**CÓMMENT** nonquote is any character other than '

$$\text{symbol} ::= \text{symch} \{ \text{symch} \}$$
$$\text{variable} ::= \text{varstart} \{ \text{alphanum} \}$$

```
varstart ::= bigletter | _
```

$$\text{integer} ::= - \text{digit} \{ \text{digit} \} \mid \text{digit} \{ \text{digit} \}$$

**string** ::= "{ sitem }"

**COMMENT** in Toy a string is equivalent to a list of character names; in Prolog-10, to a list of their ASCII codes

**sitem ::= ''' | nondquote**

**COMMENT** **nondquote** is any character other than "

```

smalletter ::= a | b | c | d | e | f | g | h | i |
                j | k | l | m | n | o | p | q | r |
                s | t | u | v | w | x | y | z

```

**bigletter** ::= A | B | C | D | E | F | G | H | I |  
J | K | L | M | N | O | P | Q | R |  
S | T | U | V | W | X | Y | Z

**digit** ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

**symch** ::= . | : | - | < | = | > | + | / |  
\* | ? | & | \$ | @ | # | \ | \

COMMENT a lone dot followed by white space is not a symch but a fullstop

**solochar** ::= , | ; | !

**token** ::= **functor** | **variable** | **integer** |  
**string** | **bracketbar**

COMMENT tokens are listed to explain note 6 below

**bracketbar** ::= ( ) | [ ] | { } | |

**comment** ::= % { **nonlineend** } **lineend**

COMMENT **lineend** is an end-of-line (linefeed) character; **nonlineend** is any other character. Toy converts line-ends to single linefeeds

**whitespace** ::= { **layoutchar** }

COMMENT **layoutchar** is blank or tab or **lineend** or any nonprintable character (in ASCII these are characters with codes = < 31)

**fullstop** ::= . **layoutchar**

#### Notes:

1. Mixed functors have not been described, but their inclusion is straightforward:

**term**<sub>N</sub> ::= **op**[ xfy,fx ],N **term**<sub>N-1</sub>

and 11 other combinations. In Toy, a mixed functor can only have one binary and one unary type, both with the same priority.

2. There are numerous ambiguous combinations of contiguous operators. This grammar does not account for them. See Section 7.4.3 (and Appendix A.3) for a rather detailed description in Prolog.
3. Not all functors can be declared as operators. Quoted names are always taken as "normal" functors.
4. In the definition of **body**, commas and semicolons need not have been actually singled out, because they are regular infix functors. The definition

**body** ::= **nonvarint**

would not, however, emphasize the most common structure of **body**.



5. The syntax of directives conforms to the convention adopted in Toy. See Section 7.4.2 for details.
6. Comments and whitespace can be freely inserted before and after a token, and cannot be inserted in the middle of a token. Remember that a comment extends till end-of-line.
7. Whitespace must be inserted between an unsigned integer and a minus which is to be treated as a functor. A minus immediately preceding a sequence of digits is taken as a part of the integer.
8. If curly brackets are not available, the usual practice is to use "decorated brackets": %( and %). This requires some care in the treatment of comments.
9. A term on input must be terminated with a full stop not embedded in a quoted name, string or comment.

## 5.2. BUILT-IN PROCEDURES: GENERAL INFORMATION

For the purposes of this chapter, built-in procedures fall into two groups. System procedures are implemented in the interpreter described in Section 7.3. Predefined procedures are written in Prolog; they belong to the user interface described in Section 7.4. Together, these two groups cover the basic set of Prolog-10 procedures. Differences and extensions are noted where appropriate but this is a description of Toy and is not intended as a replacement for the Prolog-10 manual. The procedures are roughly classified according to their purpose.

A system procedure call may fail, succeed or raise an error. Failure or success is equivalent to a failure or success of a normal procedure call. The only difference is that success is usually accompanied by a side-effect, such as writing a character, setting a switch, etc. A failing system procedure does not usually cause any side-effects (input procedures are a notable exception).

An error is raised when a system procedure detects an incorrect parameter (or parameters). If the description of a procedure mentions the form of expected parameters, parameters of unlisted forms will cause an error to be raised. There is no guarantee that the error will be raised before any actions are performed, though this is usually so.

Raising an error consists in invoking procedure *error/1*, with its single parameter instantiated to the offending system procedure call. In general, *error* behaves as if its call were present in the program instead of the



erroneous system procedure call. An explicit call to *error* is also possible. *error* is a Prolog procedure: the standard library contains a simple version which outputs a message and fails. The user can augment this procedure to his liking, possibly providing different clauses as “error handlers” for different system procedures. Redefinition of *error* requires removing it from the standard library (see Section 7.4.5)—in the present version it is protected together with the whole library. Some predefined procedures invoke *error*, and so can the user’s programs.

*error* is not in Prolog-10.

The following are conventions observed throughout this chapter. (Additional conventions or explanations appear under some group headings.)

Whenever we say that a procedure “tries to unify” we mean that it fails or succeeds depending on the outcome. Success means that unification is performed.

When we say that a procedure “tests” something, we mean that it fails or succeeds according to the result.

Acceptable parameters are indicated by conventional names listed below:

TERM—any term will do

INTEGER—an integer

VAR—a variable

NONVARINT—a non-variable, non-integer term

CALL—same as NONVARINT

ATOM—a NONVARINT without arguments

NAME—same as ATOM

CHAR—a NAME consisting of a single character

FILENAME—a NAME conforming to the implementation-dependent conventions for specifying files

CALLIST—a list (possibly empty) of CALLs

CHARLIST—a list (possibly empty) of CHARs

DIGITLIST—a CHARLIST built of digit characters

In descriptions, PAR1, PAR2 etc. stand for actual parameters in the built-in procedure call.

Note that ‘123’ is a name, and 123 an integer. 9 is the integer nine, and ‘9’ is the digit (character). The output procedures do not always distinguish between the two (*writeln* does).

Toy introduces a number of predefined operators. Some of them are used as infix or prefix procedure names. Table 5.1 is the list of predefined operators:



TABLE 5.1

Predefined Operators

Name	Type	Priority
<code>:-</code>	xfx	1200
<code>:</code>	fx	1200
<code>--&gt;</code>	xfx	1200
<code>;</code>	xfy	1100
<code>,</code>	xfy	1000
<code>not</code>	fy	900
<code>=</code>	xfx	700
<code>is</code>	xfx	700
<code>=:=</code>	xfx	700
<code>=\=</code>	xfx	700
<code>&lt;</code>	xfx	700
<code>=&lt;</code>	xfx	700
<code>&gt;</code>	xfx	700
<code>&gt;=</code>	xfx	700
<code>@&lt;</code>	xfx	700
<code>@=&lt;</code>	xfx	700
<code>@&gt;</code>	xfx	700
<code>@&gt;=</code>	xfx	700
<code>==</code>	xfx	700
<code>\==</code>	xfx	700
<code>=..</code>	xfx	700
<code>+</code>	yfx	500
<code>+</code>	fx	500
<code>-</code>	yfx	500
<code>-</code>	fx	500
<code>*</code>	yfx	400
<code>/</code>	yfx	400
<code>mod</code>	xfx	300

### 5.3. CONVENIENCE

**true**

always succeeds.

**fail**

always fails.

**not CALL**

the "not" procedure (but see Section 4.3.2!): succeeds only when the parameter fails. Defined in Prolog:

not C :- C, !, fail.

not \_.

**CALL , CALL**

the “and” procedure: succeeds only when both arguments succeed.  
Defined in Prolog:

$A, B :- A, B.$

See also the description of the cut.

**CALL ; CALL**

the “or” procedure: succeeds only if either of the parameters succeeds. Defined in Prolog:

$A; _ :- A.$   
 $_; B :- B.$

See also the description of the cut.

**check(CALL)**

succeeds only when the parameter succeeds, but instantiates no variables—only side-effects of CALL remain. Defined in Prolog:

$\text{check}(\text{Call}) :- \text{not not Call}.$

Not in Prolog-10.

**side\_effects(CALL)**

exactly equivalent to  $\text{check}(\text{Call})$ , but used when the parameter is to be executed for its side-effects rather than to test something. Not in Prolog-10.

**once(CALL)**

executes CALL deterministically. Defined in Prolog:

$\text{once}(\text{Call}) :- \text{Call}, !.$

Not in Prolog-10.

## 5.4. ARITHMETIC

In the descriptions,  $\text{div}$  stands for integer division, and  $\text{mod}$  for taking the remainder of integer division.

The following are correct invocation patterns for  $\text{sum}/3$  (not in Prolog-10).

**sum(INTEGER, INTEGER, INTEGER)**

succeeds only if  $\text{PAR1} + \text{PAR2} = \text{PAR3}$

**sum(INTEGER, INTEGER, VAR)**

succeeds after unifying  $\text{PAR3}$  with the value of  $\text{PAR1} + \text{PAR2}$

**sum(INTEGER, VAR, INTEGER)**

succeeds after unifying  $\text{PAR2}$  with the value of  $\text{PAR3} - \text{PAR1}$



`sum(VAR, INTEGER, INTEGER)`

succeeds after unifying `PAR1` with the value of `PAR3 - PAR2`

The following are correct invocation patterns for *prod/4* (not in Prolog-10).

`prod(INTEGER, INTEGER, INTEGER, INTEGER)`

succeeds only if  $PAR1 * PAR2 + PAR3 = PAR4$

`prod(INTEGER, INTEGER, INTEGER, VAR)`

succeeds after unifying `PAR4` with the value of  $PAR1 * PAR2 + PAR3$

`prod(INTEGER, INTEGER, VAR, INTEGER)`

succeeds after unifying `PAR3` with the value of  $PAR4 - PAR1 * PAR2$

`prod(INTEGER, VAR, VAR, INTEGER)`

succeeds after unifying `PAR2` with the value of  $PAR4 \text{ div } PAR1$  and `PAR3` with the value of  $PAR4 \text{ mod } PAR1$

`prod(VAR, INTEGER, VAR, INTEGER)`

like the previous one, but with `PAR1` and `PAR2` exchanged

`prod(INTEGER, VAR, INTEGER, INTEGER)`

fails if  $(PAR4 - PAR3) \text{ mod } PAR1$  is not zero; otherwise succeeds after unifying `PAR2` with the value of  $(PAR4 - PAR3) \text{ div } PAR1$

`prod(VAR, INTEGER, INTEGER, INTEGER)`

like the previous one, but with `PAR1` and `PAR2` exchanged

**TERM is TERM**

the procedure *is* assumes `PAR2` is an integer expression, i.e. a term composed of integers by means of standard arithmetic functors: + (binary and unary), - (binary and unary), \*, /, mod. The procedure fails if `PAR2` is not an integer expression. Otherwise it evaluates the expression and tries to unify the value with `PAR1`. According to Prolog-10 conventions, *is* can also evaluate a list

[ INTEGER ]

as this `INTEGER`; e.g. `55 is [55]` succeeds. (This is needed in Prolog-10 mainly for evaluating single character strings to ASCII codes.) Defined in Prolog.

## 5.5. COMPARING INTEGERS AND NAMES

`less(INTEGER, INTEGER)`

succeeds only if  $PAR1 < PAR2$ . Not in Prolog-10.

**TERM =:= TERM**

PAR1 and PAR2 are treated as integer expressions and evaluated. The procedure succeeds only if both parameters are proper integer expressions (see *is/2*) and their values are equal. Defined in Prolog.

**TERM =\= TERM**

as above, but tests whether the values are nonequal

**TERM < TERM**

as above, but tests whether the value of PAR1 is less than that of PAR2

**TERM =< TERM**

as above, but tests whether the value of PAR1 is not greater than that of PAR2

**TERM > TERM**

as above, but tests whether the value of PAR1 is greater than that of PAR2

**TERM >= TERM**

as above, but tests whether the value of PAR1 is not less than that of PAR2

**NAME @< NAME**

succeeds only when PAR1 precedes PAR2 in the lexicographic order (as defined by the underlying ASCII collating sequence).

**NAME @=< NAME**

like @<, but tests whether PAR2 does not precede PAR1. Defined in Prolog.

**NAME @> NAME**

like @<, but tests whether PAR2 precedes PAR1. Defined in Prolog.

**NAME @>= NAME**

like @<, but tests whether PAR1 does not precede PAR2. Defined in Prolog.

## 5.6. TESTING TERM EQUALITY

**TERM = TERM**

tries to unify PAR1 and PAR2. Defined in Prolog:

**X = X.**

**eqvar(VAR, VAR)**

succeeds only when the parameters are two occurrences of the same nondummy variable. Not in Prolog-10.



**TERM == TERM**

succeeds only when the parameters are two occurrences of the same term. For example, if A, B are uninstantiated,

`p( A ) == p( B )`

fails, even though

`p( A ) = p( B )`

succeeds. Defined in Prolog.

**TERM \== TERM**

succeeds only when the parameters are not two occurrences of the same term. Defined in Prolog.

## 5.7. INPUT/OUTPUT

### 5.7.1. Switching Streams

This set of procedures can be used to dynamically change the files read or written by the input/output procedures. The user's terminal is treated like any other file: its name is *user* (both for input and output); the terminal is read from and written on by default.

Ideally, one should be able to open a file with *tell* or *see*, stop using it with another *tell* or *see*, start using it from the current position after a second *tell* or *see*, and close it with *told* or *seen*. There should be no limits on the interleaving introduced by using a file in the middle of using a file in the middle etc.

The procedures are described as if this situation were real. In practice, things are very implementation-dependent. The version of Toy presented in Chapter 7 has only two input and two output streams: one for the terminal and one for a disk file in each direction. Also, Toy has no code for dealing with incorrect file names, nonexistent files and the like. All this is too dependent on the environment in which it is implemented.

**see(FILENAME)**

the specified file becomes the current input file; the terminal's name is *user*

**seeing(TERM)**

tries to unify the parameter with the name of the current input file

seen

closes the current input file; *user* becomes current. Has no effect if the current file is *user*

tell(FILENAME)

the specified file becomes the current output file; the terminal's name is *user*

telling(TERM)

tries to unify the parameter with the name of the current output file told

closes the current output file; *user* becomes current. Has no effect if the current file is *user*

### 5.7.2. Listing Control

The Toy-Prolog interpreter contains a listing switch. If the switch is on, each line read in from the current input is listed on the user's terminal: this is useful when one wants to see what is being read from a disk file.

echo

succeeds after turning the listing switch on; has no effect if the switch is already on. Not in Prolog-10.

noecho

succeeds after turning the listing switch off; has no effect if the switch is already off. Not in Prolog-10.

### 5.7.3. Terms

display(TERM)

writes the term onto the current output. The term is written in standard notation (prefix with parentheses) and identifiers are not quoted even if they normally should be. Variables are written as *\_n*, where *n* is an address. There is no guarantee that a variable will be printed as the same address in different invocations of *display*. In Prolog-10, *display* is a little different: it always writes on the user's terminal.

write(TERM)

writes the term onto the current output. The term is written according to operator declarations currently in force. No identifiers are quoted. Variables are written as *X1*, *X2* etc. Each invocation of *write* begins numbering from 1, so that e.g. the calls

```
write( X ), write( f ( Y, X ) )
```

will produce

```
X1f( X1, X2 )
```



Defined in Prolog. CAUTION: in Toy, *write* uses *numbervars* (see Section 5.15) which binds variables in the term to 'V'(N) for N = 1, 2, etc. Hence, *write* cannot output any term 'V'(INTEGER) properly.

**writeq(TERM)**

same as *write*, but quotes identifiers that are not proper words or symbols, and also those identifiers that coincide with operator names; e.g. a 3-parameter *is* would be quoted. However, a quote within a quoted name will not be doubled (this is a bug, actually). Otherwise, a term written by *writeq* can be read back by *read*.

**read(TERM)**

reads from the current input a term, terminated with a full stop. Succeeds only when PAR1 unifies with this term. Operator declarations currently in force are taken into account. Recall that a quoted name cannot be an operator. If the text on input is not a correct term, *read* prints the message

+++ Bad term on input. Text skipped:

skips and reprints the input until the first (still unprocessed) full stop, and tries to unify PAR1 with 'e r r'. (If the erroneous line does not contain a full stop, you should input one before Prolog resumes.) See the next section for behaviour on file end detecting. Defined in Prolog in terms of single-character input (see the next section).

**op(INTEGER, TERM, ATOM)**

declares an operator with PAR3—the name, PAR1—the priority ( $1 \leq \text{PAR1} \leq 1200$ , and PAR2—the type. PAR1 is usually less than 1000, to avoid conflicts with clause-constructing operators (see the table in Section 5.2); operators with lower priority take precedence over those with a higher priority. PAR2 must be a proper word or symbol. Admissible types of operators are fx, fy (unary, prefix); xf, yf (unary, postfix); xfx, xfy, yfx (binary, infix). The types fx, xf, xfx are non-associative; fy, yf, associative; xfy, right-associative; yfx, left-associative. Any other PAR2 causes an error.

If an operator declaration with this name but another priority is already in force, the procedure replaces the old declaration with the new one. If a declaration with the same name and priority exists, three possibilities arise:

- both operators are binary or both unary; the old definition is replaced;
- the old operator is unary (binary), the new—binary (unary); a mixed functor is declared;



—the old operator is mixed, the new—binary (unary); the binary (unary) type in the mixed functor declaration is replaced with PAR2.

Defined in Prolog.

delop(ATOM)

the operator declaration with the name given by PAR1 is deleted. The name should be quoted to prevent it from being treated as an (erroneous) operator with missing arguments. Defined in Prolog. Not in Prolog-10.

#### 5.7.4. Single Characters

The Toy interpreter contains a single-character input buffer called the **current character**. Initially, it contains a blank and is then refilled by each reading operation. In the presented version, each line end is treated as if it were a linefeed character (ordinal number 10, see the procedure *iseoln*). Behaviour upon detection of end-of-file depends on the current input. If the input is *user* (i.e. the terminal), Prolog is terminated; otherwise an automatic *seen* is performed and the reading operation is restarted.

The operations presented here (except *nl*) differ from those in Prolog-10. In Toy, the arguments of input/output operations are characters, and the internal buffer can be used to rescan the current input character. In Prolog-10 there is no such buffer and the arguments of the operations are *integers*, i.e. character codes. These operations could be defined as follows:

```
get0( Ord ) :- rch, lastch( Ch ), ordchr( Ord, Ch ).
```

```
get( Ord ) :- rch, skipbl, lastch( Ch ),
               ordchr( Ord, Ch ).
```

```
skip( X ) :- repeat, get0( X ), !.
```

```
put( Ord ) :- ordchr( Ord, Ch ), wch( Ch ).
```

This would not assure complete compatibility, however. Erroneous calls would be handled a little differently and so would line ends. See also the description of strings.

rch

succeeds after filling **current character** with the next character from current input (but see the introductory remarks for effects of line end or end-of-file)

skipbl

succeeds after ensuring that **current character** is a printing character



with ordinal number greater than 32. Does nothing if it already is such a character; otherwise repeatedly invokes *rch*.

**lastch(TERM)**

tries to unify its parameter with **current character**

**wch(CHAR)**

writes the character on current output (the linefeed character is interpreted as line terminator)

**nl**

terminates the current output line. Defined in Prolog:

```
:- ordchr( 10, Ch ), assert( ( nl :- wch( Ch ) ) ).
```

**rdch(TERM)**

gets the next character from current input (by invoking **rch**). Makes a copy of **current character**, treating a non-printing character (including line end) as a blank; tries to unify the copy with its parameter. Defined in Prolog.

**rdchsk(TERM)**

same as above, but preceded by a call on **skipbl**

### 5.7.5. Others

These procedures are not really concerned with input/output, but the only effect of *status* is to write something, and *ordchr* is most useful when reading or writing non-printing characters. They all are not in Prolog-10.

**ordchr(INTEGER, CHAR)**

succeeds only when PAR1 is the ordinal number (ASCII code) of PAR2

**ordchr(VAR, CHAR)**

succeeds after unifying the variable with the ordinal number of the character

**ordchr(INTEGER, VAR)**

succeeds after unifying the variable with the character whose ordinal number is the value of PAR1 mod 128

**iseoln(TERM)**

tries to unify PAR1 with the end-of-line character. Defined in Prolog:

```
:- ordchr( 10, Ch ), assert( iscoln( Ch ) ).
```

**status**

writes memory utilisation information on the current output

See also *consult/1*, *reconsult/1*, *listing/0* and *listing/1* in Section 5.11.

### 5.8. TESTING CHARACTERS

Each of these procedures fails or succeeds depending on whether its parameter is a character belonging to a particular class. They are designed to help the user interface (see Section 7.4) in reading Prolog terms, but some of them are of general utility. The procedures *iseoln/1* and *ordchr/2* (see Section 5.7.5) are also used to test characters.

**smallletter(TERM)**

tests whether the parameter is a lower case letter

**bigletter(TERM)**

tests whether the parameter is an upper case letter

**letter(TERM)**

tests whether the parameter is an upper or lower case letter

**digit(TERM)**

tests whether the parameter is a decimal digit

**alphanum(TERM)**

tests whether the parameter is a letter, a digit or an underscore character

**bracket(TERM)**

tests whether the parameter is one of the following characters:

`()[]{}`

**solochar(TERM)**

tests whether the parameter is one of the following characters:

`! , ;`

**symch(TERM)**

tests whether the parameter is one of the following characters:

`+ - * / = @ # $ % & : . ? < > \`

### 5.9. TESTING TYPES

These procedures fail or succeed depending on the form of their arguments.

**var(TERM)**

tests whether the parameter is an uninstantiated variable

**integer(TERM)**

tests whether the parameter is an integer



**nonvarint(TERM)**

tests whether the parameter is a NONVARINT (neither a variable nor an integer); not in Prolog-10

**atom(TERM)**

tests whether the parameter is an atom (a NONVARINT without arguments)

## 5.10. ACCESSING THE STRUCTURE OF TERMS

The procedures *pname* and *pnamei* are not in Prolog-10. They replace *name/2*, which is similar, but which uses lists of integers (ASCII codes) in place of our lists of characters (see Section 5.7.4).

**pname(NAME, TERM)**

builds a list of characters forming the name and tries to unify it with PAR2

**pname(VAR, CHARLIST)**

succeeds after unifying the variable with a NAME formed of the characters on the list. (Note that *pname*(X, [1, 2, 3]) binds X to the name '123', and not to the integer 123).

**pnamei(INTEGER, TERM)**

builds a list of decimal digit characters (constituting the written form of the integer) and tries to unify it with the term; the integer must not be negative.

**pnamei(VAR, DIGITLIST)**

succeeds after unifying the variable with an integer whose written form is given by the digit characters on the list. Even when the parameters are formally correct, an error may be raised if the specified integer is too large.

**functor(VAR, INTEGER, 0)**

PAR3 is the integer zero; succeeds after unifying the variable with PAR2 (this version is allowed for completeness, see below for sensible uses of *functor*)

**functor(VAR, NAME, INTEGER)**

succeeds after unifying the variable with a term whose main functor has the name and arity defined by PAR2 and PAR3, and whose arguments are different variables; PAR3 must not be negative

**functor(INTEGER, TERM, TERM)**

tries to unify PAR2 with PAR1 and PAR3 with the integer zero

**functor(NONVARINT, TERM, TERM)**

tries to unify PAR2 and PAR3 with the name and arity of the main functor in PAR1

**arg(INTEGER, NONVARINT, TERM)**

fails if the integer is smaller than 1 or greater than the arity of the main functor in PAR2. Otherwise tries to unify PAR3 with that argument of PAR2 whose number is given by PAR1.

The following are correct invocation patterns for the procedure `=..` (pronounced "univ"), which is defined in Prolog.

**VAR =.. [INTEGER]**

succeeds after unifying the variable with the integer

**VAR =.. [NAME | TERM]**

if the term is not a closed list, an error in the procedure *length/2* is raised (`=..` uses *length*). Otherwise a term with NAME as its name and TERM as its argument list is created and unified with VAR.

**INTEGER =.. TERM**

tries to unify the term with [PAR1]

**NONVARINT =.. TERM**

constructs a list, with PAR1's main functor as the head and the list of PAR1's arguments as the tail. Tries to unify the list with PAR2.

## 5.11. ACCESSING PROCEDURES

The Toy-Prolog interpreter supports *assert/3*, *retract/3* and *clause/5*. These are low-level, but quite powerful procedures (see the editor in Appendix A.4). Parameters representing clause bodies have the form of lists of calls (an empty body is []).

The Prolog library uses these low-level routines to define Prolog-10 procedures *assert/1*, *asserta/1*, *assertz/1*, *retract/1* and *clause/2*. Unlike most other built-in procedures, *retract/1* and *clause/2* are non-deterministic. Parameters representing clause bodies have the form of terms used in the external representation, i.e. sequences built with commas (an empty body is *true*).

An attempt to apply any of these procedures to system routines defined by the interpreter is treated as an erroneous call. So is an attempt to modify protected procedures. (There is a diagnostic printout which cannot be suppressed by redefining *error/1*.)

Caution: Remember the standard operator declarations listed in Section 5.2. To be safe, always enclose a clause-representing term in paren-



theses. For example,

```
assert( ( a :- b, c ) )
```

is okay, but

```
assert( a :- b, c )
```

is a call on *assert/2*. In some versions of Prolog, even

```
assert( a :- b )
```

is incorrect.

**assert(NONVARINT, CALLIST, INTEGER)**

PAR1 is treated as a clause's head, PAR2 as its body. The clause is asserted immediately after the n-th clause of this procedure (where n is PAR3 if a clause with this number exists, and the last clause's position if PAR3 is too large; if the procedure is empty or PAR3 < 1, the clause is asserted as first). Not in Prolog-10.

**retract(NAME, INTEGER, INTEGER)**

PAR2 must not be negative. PAR1 and PAR2 define the name and arity of a predicate symbol. If the associated procedure does not contain a clause whose number is given by PAR3 (the first clause has number 1), *retract* fails. If the clause does exist, it is logically removed from the procedure and *retract* succeeds. A removed clause does not disappear from storage and its active instances can still run to completion. Not in Prolog-10.

**clause(NAME, INTEGER, INTEGER, TERM, TERM)**

PAR2 must not be negative. PAR1 and PAR2 are treated as the name and arity of a predicate symbol. If the associated procedure has no clause whose number is given by PAR3 (in particular, if it is a system routine) then *clause* fails. Otherwise it tries to unify PAR4 with the head of the clause and PAR5 with its body. Not in Prolog-10.

**asserta(NONVARINT)**

treats the parameter as a clause (non-unit if its main functor is :-/2, unit otherwise). An error is raised if the first argument of a :- is not a NONVARINT. Asserts the clause at the beginning of its procedure, creating the procedure if it does not exist. Defined in Prolog.

**assertz(NONVARINT)**

same as above, but the assertion is at the end of the procedure.

**assert(NONVARINT)**

equivalent to *asserta*(PAR1).

**retract(NONVARINT)**

the parameter is treated as a clause (non-unit if its main functor is :-/2 and unit otherwise). An error is raised if the first argument of :- is not



a NONVARINT. The first matching clause is retracted, and a fail point created (see Section 5.12). On failure, the next matching clause will be retracted. Note: if PAR1 has the form *nonvarint:-var* then it matches only clauses with a single call in their bodies. Defined in Prolog.

**clause(NONVARINT, TERM)**

tries to locate the first procedure whose head matches PAR1 and whose body matches PAR2; the body of a unit clause is the term *true*. After successful unification, establishes a fail point and succeeds; the next matching clause is sought on failure. Defined in Prolog. Not in Prolog-10.

**redefine**

this procedure is needed to implement *reconsult* and should not be used directly. It modifies the effects of *assert*: if the procedure to which a clause is added is different from that affected by the last assertion, an automatic *abolish* is invoked before the *assert*. The next invocation of *redefine* restores the original situation.

**protect**

succeeds after ensuring that all procedures already defined, except those whose heads are single characters with no arguments (this restriction is imposed by a minor technical difficulty), are protected. An attempt to modify a protected procedure (by means of *assert*, *retract*, *abolish*, *consult*, *reconsult*) is treated as an erroneous invocation of the system procedure in question. (The user interface in Toy protects all its procedures.) Not in Prolog-10.

**abolish(NAME, INTEGER)**

PAR2 must not be negative. PAR1 and PAR2 are treated as the name and arity of a predicate symbol. All the clauses of this procedure are logically removed (retracted) and *abolish* succeeds.

**predefined(NAME, INTEGER)**

PAR2 must not be negative. PAR1 and PAR2 are treated as the name and arity of a predicate symbol. If the procedure associated with this symbol is a system procedure, *predefined* succeeds; otherwise it fails. Not in Prolog-10.

**consult(FILENAME)**

*sees* the named file and enters program definition mode: successive terms are read-in and stored via *assertz* (see the convention for *asserta*'s parameters) and asserted (but see *protect*). There are two exceptions: the term *end* causes the file to be closed and definition mode to be exited; terms with the unary *:-* as a main functor are treated as commands, and immediately executed. Defined in Prolog.

**reconsult(FILENAME)**

as above, but *redefine* is called at the beginning and at the end of



processing. Contiguous sequences of clauses with the same predicate symbol in their heads are treated as complete definitions of procedures and supersede previous definitions.

listing(NONVARINT)

PAR1 must be an ATOM, or a term of the form ATOM/INTEGER or a list of such terms (possibly multi-level). Each atom is treated as a procedure's name, each integer as a procedure's arity. All relevant procedures are listed on the current output. Defined in Prolog.

listing

as above, but for all defined procedures (including procedures defined in the monitor and library, but excluding built-in system procedures).

## 5.12. CONTROL

Whenever a procedure call activates a clause which is not the last clause in its procedure, we say that a fail point is associated with the call. A fail point is something to backtrack to: it saves information necessary for reestablishing the state of the computation and proceeding with the next clause.

The immediate descendants of a call *C* are the calls in the procedure which *C* activated. The immediate ancestor of a call *C* is the call which activated the procedure containing *C*. An ancestor is the immediate ancestor or an ancestor of the immediate ancestor. A descendant is defined similarly.

!

the cut procedure; succeeds after finding the nearest ancestor which is not a *call/1*, *tag/2*, *,/2* or *;/2* and removing all existing fail points associated with this ancestor and all its descendants.

repeat

an endless "generator of successes" (see Section 4.3.2). Defined in Prolog:

repeat.

repeat :- repeat.

call(CALL)

behaves exactly as if its parameter were in its place, with the exception that an incorrect parameter (an integer or uninstantiated variable) is detected at run time rather than at clause-definition time. In top-level syntax, one can use a variable instead of a predicate—this is converted to an invocation of *call*.



**halt(ATOM)**

stops the interpreter after writing the atom. Not in Prolog-10.

**stop**

stops the interpreter. Not in Prolog-10. Defined in Prolog.

The following procedures are not in Prolog-10. They are useful for error handling, but are “dirty”, and should be used sparingly.

**tag(CALL)**

this is a form of *call/1* which can be referred to by *tagfail/1*, *tagexit/2*, *tagcut/2* and *ancestor/2*. The parameter of *tag* is called a “tagged ancestor” of its descendants; it is never removed from the stack as a result of tail recursion optimisation (see Sections 6.4 and 7.1).

NOTE: a tag is recognized only when explicitly written in its clause. In particular *call(tag(C))* is equivalent to *call(call(C))*.

**ancestor(TERM)**

searches for the nearest tagged ancestor unifiable with the parameter; fails if no such ancestor is found, otherwise unifies and succeeds.

**tagcut(TERM)**

searches for the nearest tagged ancestor unifiable with the parameter. Fails if no such ancestor is found; otherwise unifies, removes all existing fail points associated with the ancestor and its descendants and succeeds.

**tagfail(TERM)**

equivalent to

*tagcut( PAR1 ), fail*

i.e. if the appropriate tagged ancestor is found, the ancestor fails immediately; otherwise *tagcut* fails.

**tagexit(TERM)**

searches for the nearest tagged ancestor unifiable with the parameter; fails if no such ancestor is found, otherwise unifies and passes control to the ancestor, which succeeds immediately.

### 5.13. DEBUGGING

The built-in debugging facilities of Toy are very primitive. There is only a wall-paper trace which displays all calls with a plus or a minus to indicate success or failure respectively (e.g. if a call fails to match two clauses and activates the third, it is shown twice with a minus and once with a plus).



A more useful—selective—tracer is listed in Appendix A.5.

There is also a switch which may cause the interpreter to output warning messages upon encountering calls on non-existent procedures. It is good practice to turn it on when debugging a program.

All these procedures are not in Prolog-10, which has a more sophisticated set of debugging aids.

**debug**

succeeds after turning tracing on (no effect if already on)

**nodebug**

succeeds after turning tracing off (no effect if already off)

**nonexistent**

succeeds after turning on warning about calls on nonexistent procedures (no effect if already on)

**nonexistent**

succeeds after turning off warning about calls on nonexistent procedures (no effect if already off)

## 5.14. GRAMMAR PROCESSING

**phrase(CALL, TERM)**

treats CALL as a nonterminal symbol of a grammar rule, schematically

nt( ARG1, ..., ARGn ),

and initiates grammar processing—with this initial symbol—by calling

nt( TERM, [], ARG1, ..., ARGn )

Defined in Prolog (see Section 4.2.6).

## 5.15. MISCELLANEOUS

**length(NONVARINT, TERM)**

PAR1 must be a closed list. Computes the length of this list and tries to unify the resulting integer with PAR2. Defined in Prolog.

**isclosedlist(TERM)**

succeeds only when the term is a closed list. Defined in Prolog. Not in Prolog-10.

**numbervars(TERM, INTEGER, TERM)**

instantiates PAR1's variables as 'V'(i), 'V'(i+1), ..., 'V'(j) where i, i+1, ..., j are consecutive integers, and i is the value of PAR2. Variables bound together are of course instantiated as the same 'V'(k). As a result, PAR1 becomes ground (obviously, this is undone upon backtracking). Then the procedure tries to unify PAR3 with j+1. For example, the call

```
numbervars( [ X, Y, X ], 6, Next )
```

where Next is uninstantiated, will instantiate

```
X ← 'V'(6), Y ← 'V'(7), Next ← 8
```

The call

```
numbervars( [ X, Y, X ], 6, not_a_number )
```

will fail.

**member(TERM, TERM)**

establishes the relationship: PAR1 is a member of the list PAR2.

Defined in Prolog:

```
member( X, [ X | Y ] ).
```

```
member( X, [ _ | Y ] ) :- member( X, Y ).
```

**bagof(TERM, CALL, TERM)**

tries to unify PAR3 with the list of PAR1's instantiations after all possible computations of PAR2 (see Section 4.2.4 for details). Prolog-10 has a more sophisticated version of this procedure. Defined in Prolog.



---

## **6 PRINCIPLES OF PROLOG IMPLEMENTATION**

---

### **6.1. INTRODUCTION**

This chapter is but a bird's-eye view on implementation techniques specific to Prolog. We assume you know how conventional block structure languages are implemented: a competent programmer could hardly escape learning these things. The discussion is kept at a level free of representation details. Chapter 7 provides a rather detailed and complete case study of one of the many ways in which the basic principles can be applied in practice.

Two topics are missing: compilation and garbage collection. To compile Prolog programs is to apply the general principles in such a way that a program is executed particularly efficiently. This is done partly by taking advantage of the underlying machine (e.g. by using machine code instead of a more compact representation of programs, trading speed for memory) and partly by performing special case analysis to detect operations which can be simplified (e.g. unification with a variable which is known to be uninstantiated). We decided that compilation is beyond the scope of this book (which already discusses implementation issues more thoroughly than the usual introduction to a programming language). The problem and techniques of garbage collection are well known, and are best studied independently of a particular programming language (though you will find that in Prolog one has to do with one of the harder variants of the problem).

## 6.2. REPRESENTATION OF TERMS

If we disregard the possibility of forming cyclic structures (see Section 1.2.3), we can see that all terms are directed acyclic graphs (DAGs). They are not necessarily trees, because different branches can converge to a common component: in linear notation we express this phenomenon by repetition, as in  $t(p(X), q(p(X), Y))$ .

In a Prolog program, several identical occurrences of a term within a single clause denote the same object. Properly speaking, this is not an object but a descriptor, or template. At execution time, it corresponds to different objects in different instances of the clause. In this and the next chapter we shall reserve the unadorned word “term” for **term instances**. Terms written in a program will be referred to as **term descriptions**. A description can have several occurrences; similarly, an instance can have several parents in a DAG.

There are many possible representations of a DAG. For our present purposes they are all equivalent, provided that it is possible to distinguish nodes corresponding to Prolog variables. On a more abstract level, however, two very different methods are used to implement term instances. Accordingly, all existing implementations of Prolog can be classified as either Structure Sharing or Non-Structure Sharing (NSS).

In principle, to form a new term instance in a Non-Structure Sharing system, one must create a new DAG. We are talking about creating new instances that correspond to term descriptions (present in the program text, or in clauses asserted after having been constructed by a program); creation of new terms as a result of unification is different. Variables are bound by being associated with pointers directed at their instantiations. These pointers are invisible, i.e. automatically dereferenced, whenever the DAG is traversed. Figure 6.1 illustrates—in a representation-independent manner—two terms, before and after unification.

A Structure Sharing system takes advantage of the fact that different instances of the same term differ only in their variable bindings. Whereas two instances of

$$t(p(X), q(p(X), Y))$$

can be

$$t(p(c), q(p(c), d)) \quad \text{and} \\ t(p(r(a)), q(p(r(a)), r(a)))$$

respectively, their general structure remains the same. The main functor must be a **t** of two arguments; **t**'s first argument must also be the first



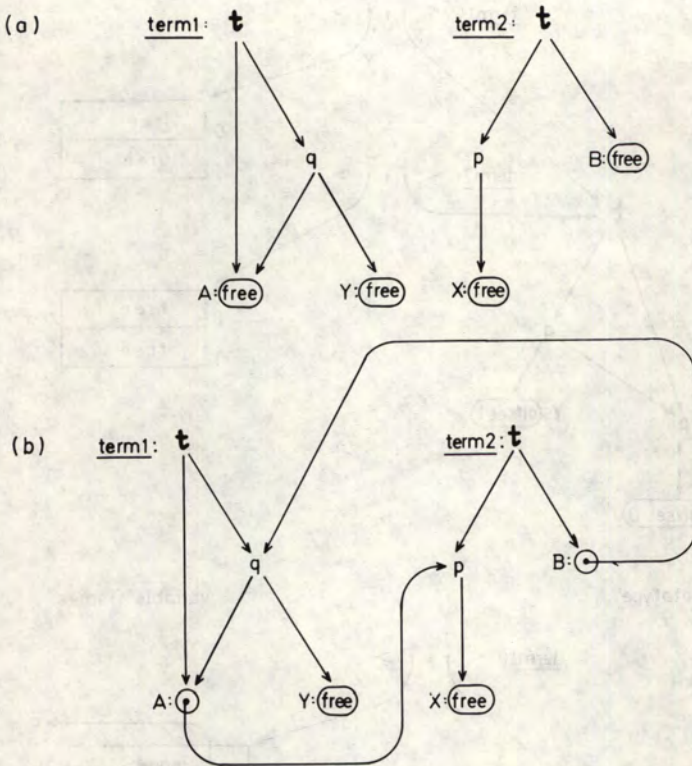


FIG. 6.1 The Non-Structure Sharing representation of terms: (a)  $t(A, q(A, Y))$  and  $t(p(X), B)$  before unification. (b)  $t(A, q(A, Y))$  and  $t(p(X), B)$  instantiated to  $t(p(X), q(p(X), Y))$  after unification.

argument of the two-argument  $q$  which is  $t$ 's second argument; and so on. Consequently, all instances of the term may share this structural information, if only care is taken to let them have different variables. This is easily achieved by associating each instance with a different **variable frame**: a chunk of storage holding variable instances. The internal representation of a term description—we shall call it **prototype**—is a DAG in which each variable node is represented by information about the offset of the variable's location in a variable frame. All terms—including variable bindings—are now represented not by single pointers, but by two-pointer **term handles**<sup>1</sup>

< prototype, variable frame >

<sup>1</sup> Another terminology, introduced by Warren (1977a), is to call prototypes **skeletons**, and handles **molecules**. We do not like the mixed metaphor.

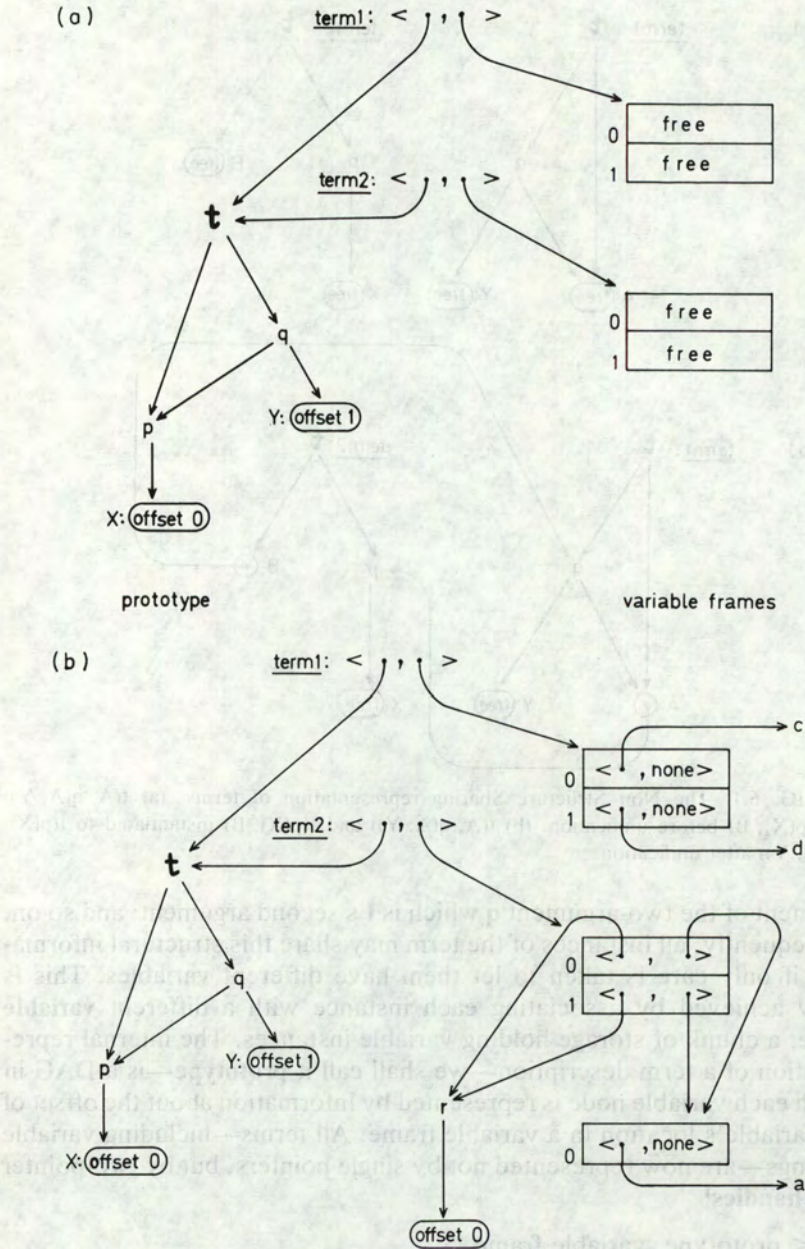


FIG. 6.2 The Structure Sharing representation of terms: (a) Two instances of  $t(p(X), q(p(X), Y))$ , both sharing the same prototype. (b)  $term1$  instantiated to  $t(p(c), q(p(c), d))$  and  $term2$  instantiated to  $t(p(r(a)), q(p(r(a)), r(a)))$ .



Figure 6.2 illustrates the principle of Structure Sharing. Figure 6.3 corresponds to Fig. 6.1. If we find general DAGs less convenient than trees, Structure Sharing makes it easy to employ trees by providing implicit links to variables from all occurrence sites. This is shown in Fig. 6.4.

Inside a clause, different occurrences of the same variable description can appear within different term descriptions. There is the problem of

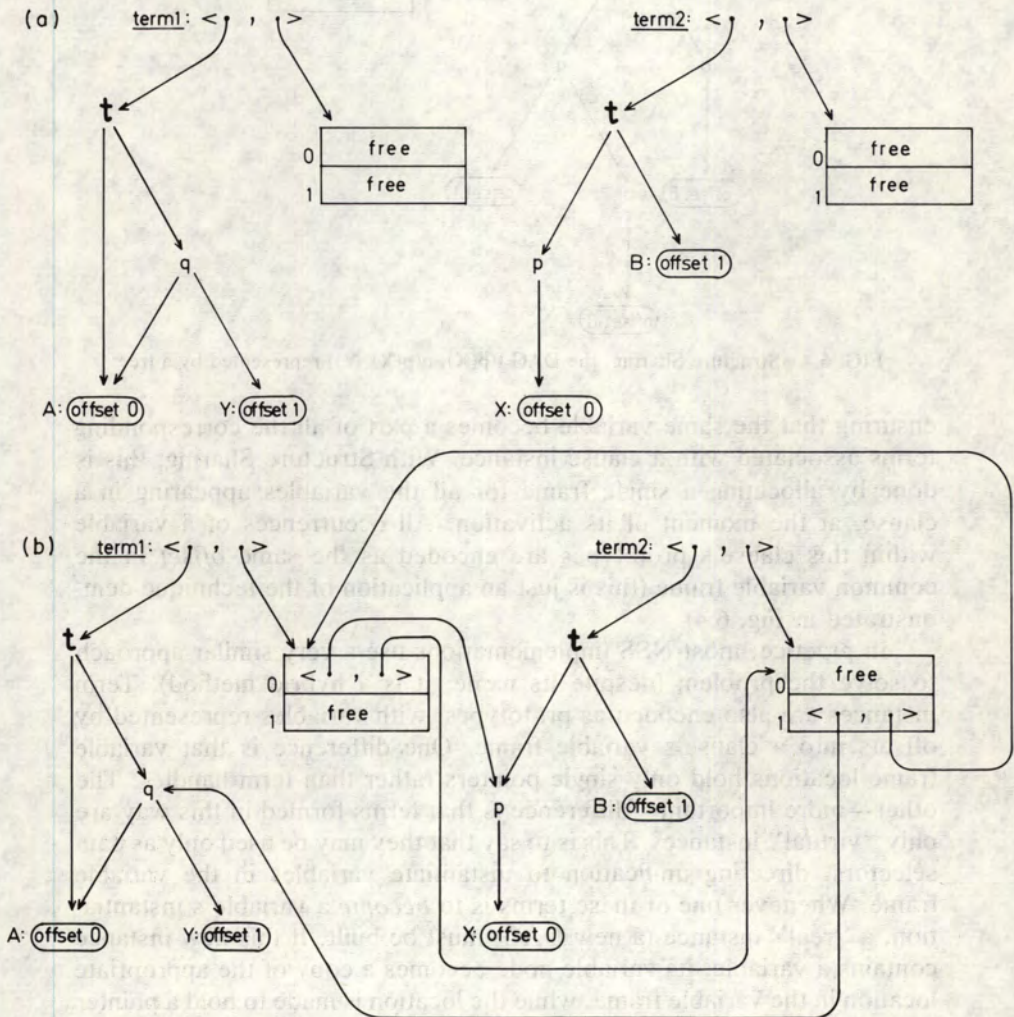


FIG. 6.3 The Structure Sharing representation of terms: (a)  $t(A, q(A, Y))$  and  $t(p(X), B)$  before unification. (b)  $t(A, q(A, Y))$  and  $t(p(X), B)$  instantiated to  $t(p(X), q(p(X), Y))$  after unification.

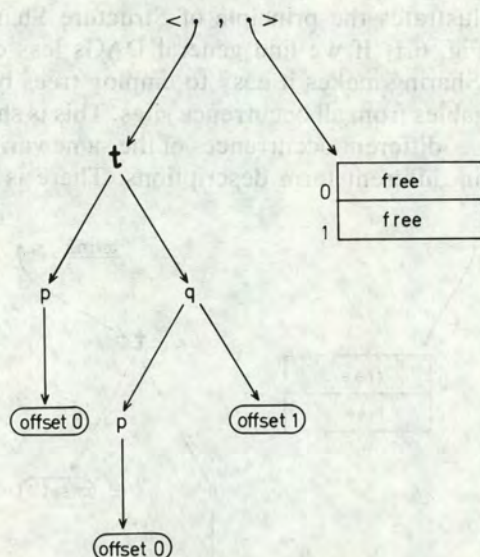


FIG. 6.4 Structure Sharing: the DAG  $t(p(X), q(p(X), Y))$  represented by a tree.

ensuring that the same variable becomes a part of all the corresponding terms associated with a clause instance. With Structure Sharing, this is done by allocating a single frame for all the variables appearing in a clause, at the moment of its activation. All occurrences of a variable within this clause's prototypes are encoded as the same *offset* in the common variable frame (this is just an application of the technique demonstrated in Fig. 6.4).

In practice, most NSS implementations use a very similar approach to solve the problem (despite its name, it is a hybrid method). Term instances are also encoded as prototypes, with variables represented by offsets into a clause's variable frame. One difference is that variable frame locations hold only single pointers rather than term handles. The other—more important—difference is that terms formed in this way are only “virtual” instances. This is to say that they may be used only as data selectors, directing unification to instantiate variables in the variable frame. Whenever one of these terms is to *become* a variable's instantiation, a “real” instance (a new DAG) must be built. If this new instance contains a variable, its variable node becomes a copy of the appropriate location in the variable frame, while the location is made to hold a pointer to the node. This ensures that all future references to the variable will end up in the node.

The process is shown in Fig. 6.5. Note that here, too, prototypes can



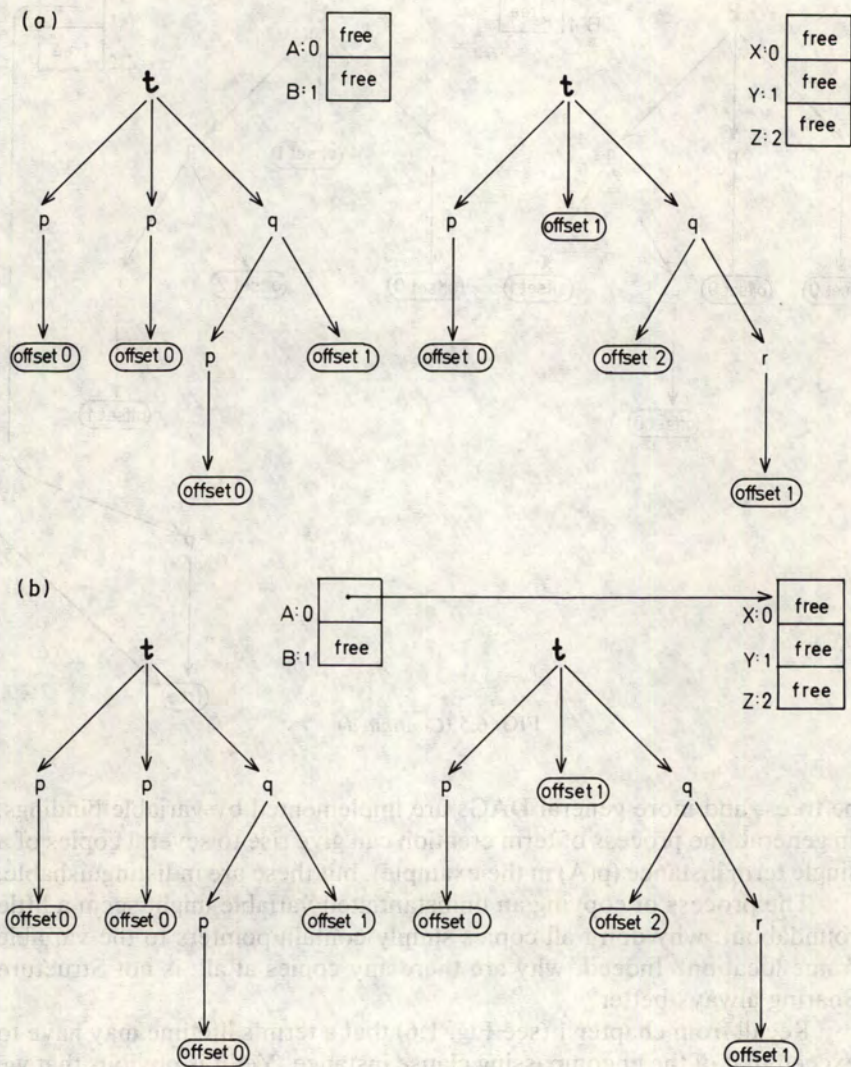


FIG. 6.5 "Virtual" and "real" instances in Non-Structure Sharing: (a)  $\text{proc}(t(p(A), p(A), q(p(A), B)))$  is called with  $\text{proc}(t(p(X), Y, q(Z, r(Y))))$ —both terms are "virtual" before unification. (b) The first occurrence of  $p(A)$  acts as a selector— $A$  is bound to  $X$ . (c) The second occurrence of  $p(A)$  acts as a constructor— $Y$  is bound to its copy (a "real" instance). (d)  $q(p(A), B)$  and  $q(Z, r(Y))$  both act as selectors, but  $p(A)$  and  $r(Y)$  are constructors—both terms are now  $t(p(X), p(X), q(p(X), r(p(X))))$ , represented by a mixture of "real" and "virtual" instances. (continued)

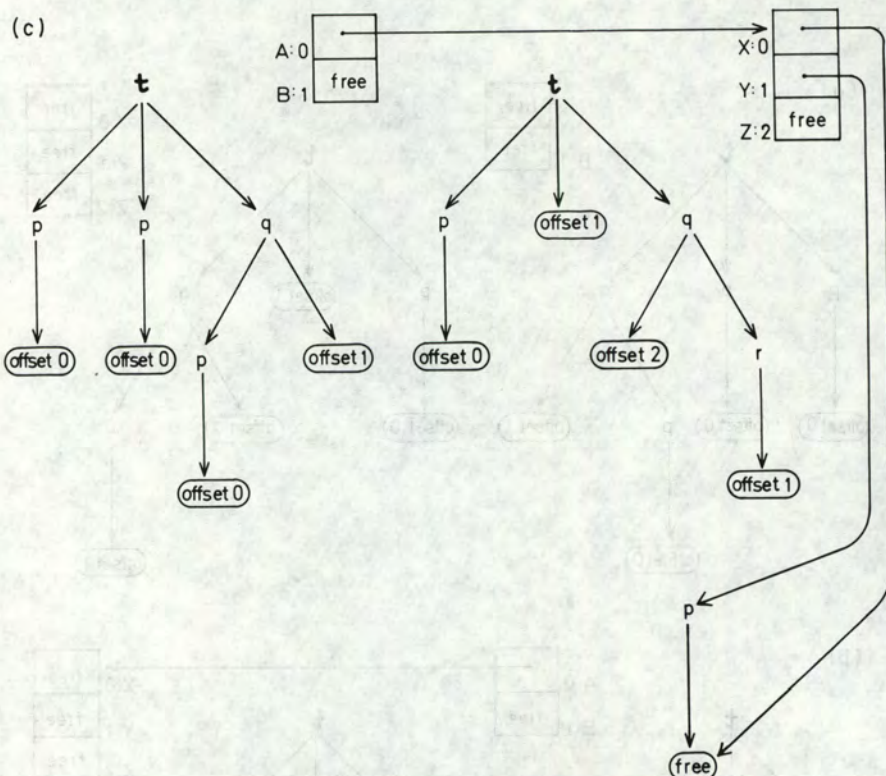


FIG. 6.5 (Continued)

be trees, and more general DAGs are implemented by variable bindings. In general, the process of term creation can give rise to several copies of a single term instance ( $p(A)$  in the example), but these are indistinguishable.

The process of copying an uninstantiated variable might seem a little roundabout: why don't all copies simply contain pointers to the variable frame location? Indeed, why are there any copies at all: is not Structure Sharing always better?

Recall from chapter 1 (see Fig. 1.6) that a term's lifetime may have to exceed that of the encompassing clause instance. Yet it is obvious that we would like to regard a variable frame—which is created when a clause is activated—as a part of the clause's activation record. If we are careful to represent variable-to-variable bindings so that younger variables point at older ones rather than the other way round, *and* if term copies contain no pointers into variable frames, then there is no risk of leaving dangling







that one may need to alleviate that by introducing artificial failures in a few well-chosen places (see Section 4.3.2).

In the simplest form of Structure Sharing, a variable frame is an integral part of the representation of a number of term instances, and cannot—in principle—be deallocated so long as any of these terms is accessible. It must be allocated on a variable stack, which closely resembles the copy stack of NSS (except that a garbage collector, if required, is harder to implement). The activation stack is smaller, as it only holds control information.

The most important advantage of NSS is that retention past the moment of procedure termination concerns only those terms which become variable instantiations. With simple Structure Sharing, on the other hand, all terms are retained. As it turns out, terms are often used as selectors rather than constructors, and clauses frequently propel a computation along without creating many long-lived objects. The copy stack is therefore usually smaller than the variable stack, and the effects of memory requirements being a function of time are much less pronounced with NSS.

Starting with DECProlog-10, many Structure Sharing implementations take advantage of the difference between terms which must live longer than their clauses and those which need not. As a clause is read in, it is analysed to detect variables which cannot, under any circumstances, be used to form instantiations of variables outside the clause. These are classified as **local** variables, whereas the others are called **global**. The variable names are all local to the clause, of course—the terminology is to convey that global variables are long-lived, while local variables may be allocated (and deallocated) with the clause's activation frame. The activation stack is accordingly referred to as the **local stack**, and the **global stack** holds global variables.

A simple, though not necessarily the most subtle, classification criterion is whether a variable appears inside a term (i.e. is not only a procedure's parameter). For example, in

$$a(X, f(Y)) :- b(X, g(Z)).$$

we find that *X* is local. The rule about directing variable-to-variable references towards the bottom of the stack suffices to ensure that its deallocation will not leave dangling pointers. The variable *Y* is obviously global, as the clause can "export" it after having been activated by

$$a(\text{Something}, \text{Variable}).$$

The status of *Z* is uncertain. It can be bound to a variable in **b**, but we are really interested only in those outside variables which outlive **a**. If the



body of **a** were

$b(g(Z))$

then  $Z$  could conceivably be classified as local (according to our experience, though, allowing such cases could complicate the implementation). But as the clause stands, we need to analyse **b** (assuming it will not subsequently be modified!) to check whether  $g(Z)$  can be made an instantiation of a variable to which  $X$  is bound. For example, with **b** defined as

$b(V, V)$

the call

$a(P, Q)$

would instantiate  $P \leftarrow g(Z)$  and  $Q \leftarrow f(Y)$ —both  $Y$  and  $Z$  would be “exported.” Variables that do not appear in terms can only be used to carry information around the clause; it is safest to assume that all others will be used to form structures.

This assumption does not yet allow Structure Sharing to be really competitive with NSS. To achieve this, we must declare our intentions by providing so-called **mode declarations**. In Prolog-10 one writes

$:- \text{mode member}(?, +).$

to inform Prolog that the second parameter of *member* will never be a variable, though its first parameter might be one. This means that the procedure

$\text{member}(E, [E | L]).$

$\text{member}(E, [X | L]) :- \text{member}(E, L).$

will not be invoked as a generator of lists, so the compound terms will only be used as selectors and all the variables—even those global by the general criterion—can be classified as local<sup>3</sup>.

Providing mode declarations may seem a nuisance, but they are good documentation (and are *not* compulsory). The declarations are static and must necessarily be less informative than the dynamic special-case analysis of NSS. In common cases, however, the difference is not detectable and this form of Structure Sharing is, in fact, as good as NSS with regard to memory utilisation. This does not mean that the two behave identically. Programs can be written which make any one method almost arbitrarily worse than the other (how would you go about devising such a program?).

<sup>3</sup> A compiler can also use this information to generate faster code.



Structure Sharing tends to be faster, but it is more complicated. There is the problem of analysing clauses, utilising mode declarations and manipulating term handles instead of single pointers. Moreover, system routines such as *clause* are harder to write because a clause can contain references to local variables and its instance is not therefore a correct term. If you want to write a simple memory-efficient interpreter, use Non-Structure Sharing.

### 6.3. CONTROL

One of the keys to the success of a Prolog implementation is the efficiency of backtracking. Whenever a fail point is established (see Section 1.3.2), the computation's state must be saved, so that it can be restored upon failure. Both the saving and the restoration of a state are frequent events, which must take place as rapidly as possible.

The state of a computation can be reduced to the contents of the control stack and the heap<sup>4</sup>. Obviously, Prolog's special requirements rule out checkpointing (i.e. dumping memory contents) as a means of saving the state. Logging (i.e. recording changes made to the state) is a more hopeful technique, as differences between successive states of interest are usually minute in comparison to the amount of information contained in a state. The technique is particularly suitable—and universally used—for dealing with the evolution of variable instantiations. Only uninstantiated variables can be modified, so the old value need not be remembered and it is enough to record a modified variable's address.

While logging is also a viable method of handling activation record traffic on the control stack, it would not be able to take advantage of the disciplined manner in which procedure instances are created and destroyed. A better method, well known since the appearance of (Bobrow and Wegbreit 1973), can roughly be described as using the log itself to define a new state.

As a fail point is established, a **fail point record** is pushed onto a special stack. (We are interested in a conceptual description. In practice, this stack is often implemented by a chain of pointers threaded through activation records.) The fail point record stores information about current sizes of memory areas and a pointer to the list of untried clauses likely to match the current call. In other words, it contains information essential to Prolog's ability to recommence computations from this fail point. To

<sup>4</sup> The generic terms are meant to emphasize that this discussion is valid for Structure Sharing and NSS alike.



make this information *sufficient*, stack and heap areas below the levels indicated by a fail point record are treated as **frozen**, i.e. under special protection.

Binding a frozen variable is allowed, but must be logged by pushing its address onto a fourth stack, called the **trail** (its size is also remembered in a fail point record). The control stack, however, is frozen quite literally. Whenever a terminating procedure would cause control to be returned to an activation record (AR) within the frozen area, a *copy* of the AR is created just above the protected part of the stack. The copy defines the current procedure's environment: an ancestor link provides access to the frozen AR of the procedure's caller. To avoid copying that part of an AR which contains variables<sup>5</sup>, *the variables of a clause are associated with the AR of its caller rather than with its own AR*. An AR's copy will be used to perform a new call: the original describes the previous call, so its variables are irrelevant. All this is illustrated in Fig. 6.6.

With these precautions, backtracking consists in undoing bindings made after creating the most recent fail point record FR (a simple matter of resetting locations referenced in the top-most fragment of the trail), popping all stacks to the levels indicated by FR, grabbing the untried clause list and popping FR itself. This is rapid enough; the unescapable penalty is that of maintaining (several copies of) frozen substacks which would normally disappear with the shortening of call chains. One of the reasons why judicious use of the cut is so important (see Section 4.3.1) is that it allows Prolog to reclaim stack storage. To invoke the cut is to pop a number of fail point records, thereby unfreezing areas of memory.

## 6.4. TAIL RECURSION OPTIMISATION

Many programming tasks are inherently iterative. For example, to test whether an item is present in a list, we must look at successive elements until either the list is exhausted or the item is found. But in Prolog we can only define *member* as a recursive procedure. Recursion is more expensive than iteration in that it requires not only time but also stack storage which grows linearly with the number of turns. Storage is often a scarce resource, and it would be very unsatisfactory if each decision to traverse a list had to be accompanied by speculations about the potential length of the list. In Prolog, using recursion instead of iteration is all the more serious because the stack may subsequently have to be frozen.

<sup>5</sup> Local variables of Structure Sharing, "virtual" variable instances of NSS.



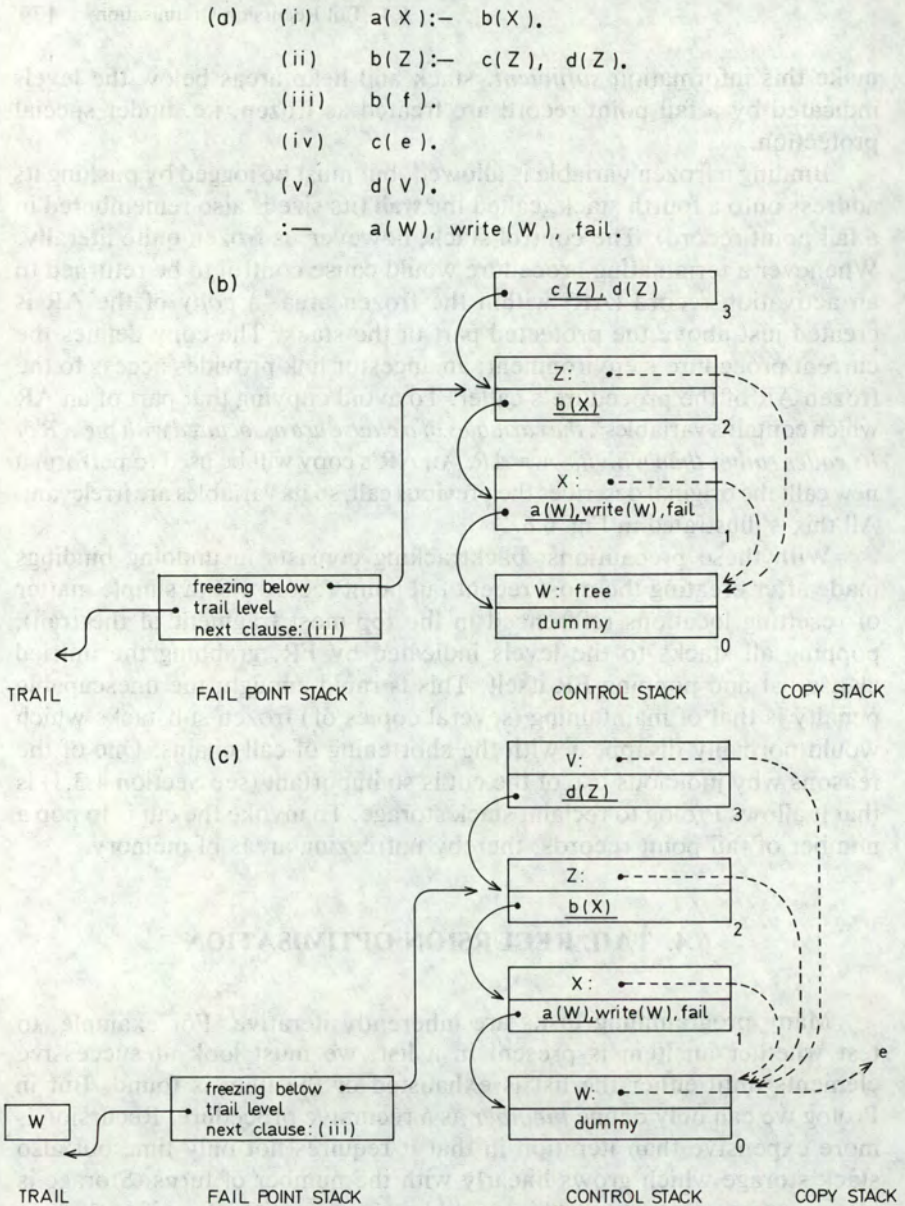


FIG. 6.6 Control stack management: (a) The example program. (b) Clause (iv) is invoked. (Solid lines in control stack are the ancestor links, dotted lines are variable bindings. Active calls are underscored, remaining calls in each clause are also shown. The model is NSS.) (c) One step later,  $d$  is ready to return. (d) After returning from  $d$  and  $b$ . (Frame 3 is a copy of 1, executing the next call. The variable  $Z$  was destroyed when the stack was popped—it was just above the freezing level.) (e) After failure, before invocation of clause (iii). (f) Clause (iii) and (i) terminated, directive in control.



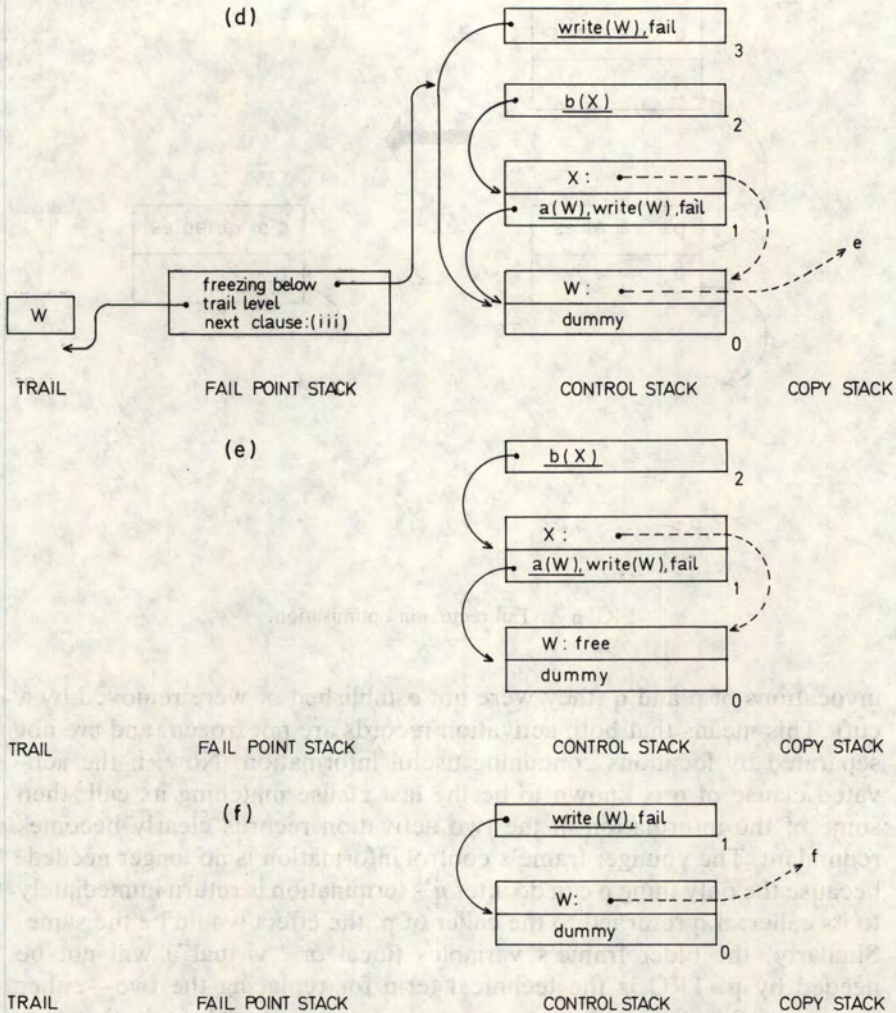


FIG. 6.6 (Continued)

Tail recursion optimisation (TRO) is the technique of replacing some forms of recursion with iteration. Despite its name, it is also useful in situations where there is no direct recursion, or even no recursion at all—just a long chain of procedure calls.

The general idea is illustrated in Fig. 6.7. Assume that **q** is the last call in **p** and that **p** is deterministic, i.e. there are no fail points between the

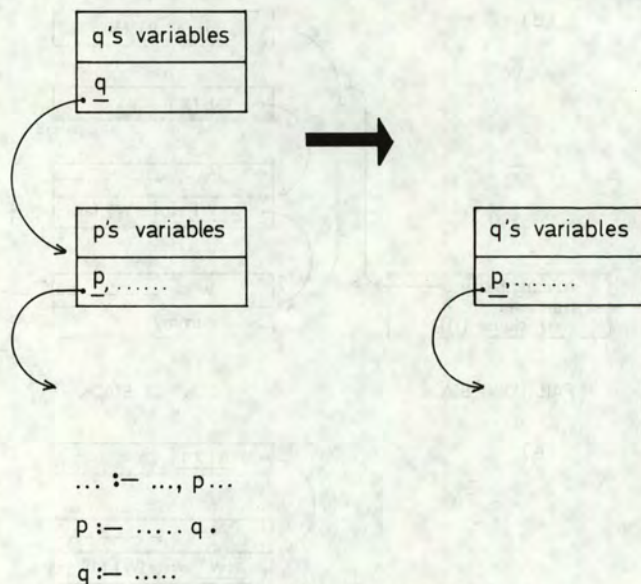


FIG. 6.7 Tail recursion optimisation.

invocations of **p** and **q** (they were not established or were removed by a cut). This means that both activation records are not frozen, and are not separated by locations containing useful information. Now, if the activated clause of **q** is known to be the last clause matching its call, then some of the information in the two activation records clearly becomes redundant. The younger frame's control information is no longer needed, because the only thing **p** can do after **q**'s termination is return immediately to its caller: if **q** returned to the caller of **p**, the effect would be the same. Similarly, the older frame's variables (local or "virtual") will not be needed by **p**. TRO is the technical term for replacing the two—either during or after **q**'s invocation—by one activation record, with **q**'s variables and with control information needed to exit from **p**. If **q** is the same as **p** (or contains a tail recursive call on **p**, or the like), many calls may be executed without increasing the size of the control stack. (The heap may grow, though, if the computation constructs some long-lived objects.)

Several methods of implementing TRO are described in the literature, and we shall not discuss them here. An important feature of some of them is that they allow **delayed** TRO, i.e. merging of our two activation records after **q** performs a cut, even though its initial invocation is not deterministic. In Section 7.3.4 you will find one such method, which we favour for its simplicity.



## 6.5. BIBLIOGRAPHIC NOTES

The idea of structure sharing comes from Boyer and Moore (1972). It was used in the original Marseilles interpreter (Battani and Méloni 1973, Roussel 1975), which, actually, was preceded by an earlier, experimental version (Colmerauer *et al.* 1972). That interpreter did not have anything like fail point records. Though variables were allocated on a separate stack, control frames were also—as a rule—popped only on backtracking. Classification of variables into local and global was introduced with the DECProlog compiler. Warren (1977a) is the original reference, see also Warren *et al.* (1977) and Warren (1980b). A preliminary report on the first NSS implementation is Bruynooghe (1976).

The idea of tail recursion optimisation is well known. Bruynooghe was the first to use TRO in Prolog, while Warren used a different method as an afterthought; see Warren (1980a).

A good detailed explanation of the implementation principles is Bruynooghe (1982b). It stresses both the similarity of structure sharing to conventional handling of procedure instances and the similarity of Prolog's control structures to a proof tree. Van Emden (1982) contains a disciplined derivation of the control algorithm, starting from search-tree traversal.

Most implementations merge fail point records and control frames into a single type of record. To our knowledge, they were first separated in Donz (1979), an early approach to global optimisation, where they were talked of as the and-nodes and or-nodes of a search tree. We like the separation because it brings to light the fact that backtracking is implemented almost exactly as proposed—in a more general setting—in Bobrow and Wegbreit (1973), the classic paper on implementation of unconventional control structures.

A comparison of NSS and structure sharing can be found in Mellish (1982), with some comments in Bruynooghe (1982b).

Mellish (1981) is an early approach to automatic production of mode declarations by means of global flow analysis. Other papers concerned with global analysis, though not for the sake of efficiency, are Bruynooghe (1982a) and Mycroft and O'Keefe (1983).

At the time of this writing we know of two new compilers being developed. The references are Bowen *et al.* (1983) and Ballieu (1983).

See also Section 2.5 for references on Prolog implementations with coroutining and parallelism.

As a point of interest, we shall mention two papers describing implementations of Prolog done by embedding it in another programming language: Lisp (Komorowski 1982) or POP-11 (Mellish and Hardy 1983).



---

## 7 TOY: AN EXERCISE IN IMPLEMENTATION

---

### 7.1. INTRODUCTION

This chapter is a case study of Toy—a simple but fairly complete implementation of Prolog. Only the most important (or least obvious) information is presented here, and it should be read *together with the source texts* available on the diskette enclosed with this book (some of these are listed in the appendices).

While designing Toy, we attempted to strike a compromise between several conflicting goals. We wanted to write:

- A clean, readable interpreter which you could find useful for “getting a feel” of what is involved in implementing a “life-size” Prolog system;
- A usable interpreter, which we could use to test all the programming examples in this book (our extant implementations were quite incompatible with Prolog-10) and which you might use to experiment with Prolog if you have a lot of time but no access to a machine running one of the commercially available Prolog systems;
- A large fragment of the implementation in Prolog itself, to provide a sizable example of using the language for solving well-known but not completely trivial programming tasks at a relatively low level;
- An interpreter which, though useful, would have little commercial value.

We decided to use Pascal, because it is easy to read, well known and generally available. The program is not written to be very efficient: concern for readability and conciseness almost always prevailed. It is not particularly short and elegant either, as we wanted it to support a



fairly complete version of Prolog modelled after the Prolog-10 dialect. There are two principal reasons why we call it *Toy*:

- The user interface is written in Prolog, and this makes it rather slow;
- There is no garbage collector, and moreover, partitioning storage into several disjoint fixed-length areas makes it easier to encounter a memory overflow condition.

If you decide to use *Toy*, you will quickly find that the time taken to read and write terms requires some patience. We had to rewrite *read*, *write* and *op* in Pascal for our purposes, and it is but a moderately difficult task. A rather straightforward implementation resulted in another 1000 lines of code, but a lot of it is dedicated to handling mixed functors (see Section 7.4.3).

We used *Toy* on two minicomputers: a PDP 11/40 look-alike running RSX11M, and a Polish computer called Mera 400. The PDP has an address space of 64KB; we used it to bring the system up, but it was a tight squeeze. You might do better with a P-code system rather than with a native-code compiler of Pascal, such as the one we had to use. The Mera had a 128KB address space and a fairly good native-code compiler (but with no attempt at global optimisation): we could easily load and execute both the whole Prolog interface and programs such as WARPLAN or Toy-Sequel (see Chapter 8). We tested all our programs and had quite a bit of memory to spare, running in a 104KB space.

The original implementation was subsequently ported (almost painlessly!) into Berkeley Pascal (on the VAX/780 running 4.2 BSD UNIX) and into TURBO Pascal (on the IBM PC running MS-DOS 2.10). You can find the TURBO version on the diskette enclosed with this book. Files READ.ME, CONTENTS, INSTALL and TURBO.PAT contain general information about the diskette and the implementation. The latter file summarizes changes introduced into the original Mera Pascal which was listed in the hardcover edition of this book.

Feel free to run *Toy* and play with it, but remember it is copyrighted. No version of this implementation may be used or distributed for gain, all listings must contain our copyright notice, and the heading produced by *status* (see Section 5.7.5) must contain the texts “Toy-Prolog” and “IIUW Warszawa”. Other than that, you are welcome to modify it, give it to friends, etc. If you have any comment to make, we shall be happy to hear from you.

## 7.2. GENERAL INFORMATION

*Toy* is a Non-Structure Sharing interpreter (see Chapter 6). The program written in Pascal supports a limited syntax, which we shall call *Toy-*



Prolog, and only a subset of the usual system (built-in) procedures. The full user interface and library is implemented in Prolog (see Section 7.4)—this approach was taken in the original Marseilles implementation, and in a number of implementations since. A short program called the “bootstrapper” (see Section 7.4.1), written in Toy-Prolog, is used to translate into Toy-Prolog other parts of the user interface, which are written in a slightly restricted form of the usual syntax. Next, various interface programs can be loaded during initialization (see Section 7.3.6).

A Prolog program called the “monitor” supports an interactive programming regime (see Section 7.4.2). Full Prolog-10 syntax can be used (see sections 7.4.3–7.4.5). A program called the “translator” can be used to convert Prolog-10 programs into Toy-Prolog (see Section 7.4.6). The translator shares most of the monitor’s routines. It can be used for large (interactively debugged) programs which are to be loaded quickly, without repeated syntactic analysis by the rather slow parser in the monitor. See Appendix A.4 for a few examples of such programs.

We shall finish this section with an example of Toy-Prolog syntax. There is no point in providing a precise description of this language, as it is very simple and the recursive-descent parser (a fragment called the **READER**, see file **READER.PAS** on the diskette) is so straightforward that it can easily be used to resolve all doubts. Our example is

```
p([a, [b, c], d | X], Y) :- q(Y, X), r(s(Y), _).
:- p(Z, (t :- u, v)).
```

To make it directly acceptable to the **READER**, we write

```
p(a.(b.c.[ ]).d.:0, :1) : q(:1, :0) . r(s(:1), _ ) . []
: p(:0, ':-'(t, ':(u, v))) . []#
```

See Appendix A.2 for further examples. The syntax is not nice, but is very close to the internal representation of clauses.

### 7.3. THE TOY-PROLOG INTERPRETER

#### 7.3.1. The Principal Storage Areas

Toy uses several disjoint areas of memory for its data structures (see Fig. 7.1). They are listed below.

- CT (character table), used to store strings: print names of Prolog functors and predicate symbols;
- AT (atom table), used to store atoms. In this chapter “atom” does *not* denote a functor with no arguments. It is the generic name of a record



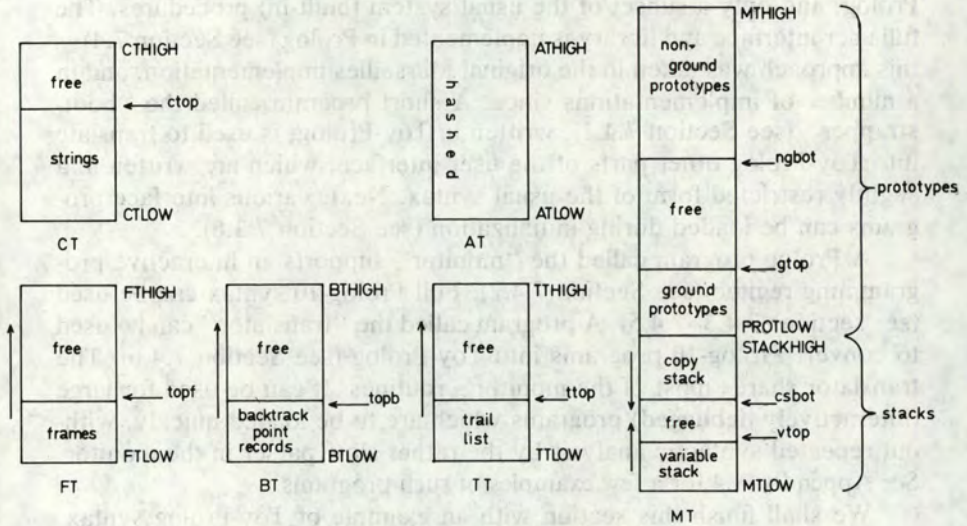


FIG. 7.1 The main data areas.

containing useful information about a symbol (a functor or predicate symbol in our case);

- MT (main table), used to store term instances and prototypes. There are two subareas here:
  - Prototype storage, which is further divided into disjoint storage areas for ground (variable-free) prototypes and for those that contain variables. The classification is important because a ground prototype can be used to represent all its instances, and need not be copied onto the copy stack;
  - Stack storage, which is further divided into disjoint areas for the copy stack and the variable stack (the variable stack holds variables from activation records: Pascal's type mechanism made it more convenient to keep control information from activation records in a separate table FT);
- FT (frame table), used as the activation record stack (but variables are stacked in MT);
- BT (backtrack table), used as the fail-point stack (here called back-track-point stack—we just needed a different letter to label the table);
- TT (trail table), used as the trail stack;
- Pascal's heap, used to store procedure descriptors;
- Pascal's stack, used for recursion in unification and term-copying operations.

In what follows, we shall use the word **pointer**, or **address**, to denote both Pascal pointers and indices into the tables.



### 7.3.2. The Dictionary: Atoms and Procedure Descriptions

The character and atom tables form the **dictionary**: a data structure used primarily as an aid in translating between the external and internal forms of Prolog terms and clauses. It also supports access to procedures, making it easier to implement variable calls and clause manipulation.

An atom is a record containing information about a functor and/or a predicate symbol. The difference between a term and a procedure depends only on context and is not always recognized. Predicate symbols are denoted by functors when clauses are treated as terms (e.g. in *assert*); conversely, a functor may be used to invoke a procedure (as in *call*).

The attributes of an atom are

- Its print name (a pointer to a string in CT);
- Its arity;
- The procedure of this name and arity (a pointer to a procedure descriptor, or **nil**).

Atoms are accessed through direct pointers or through a hashing procedure. Direct pointers are present in the representation of terms (including clauses; see the next section). The pointers are used for

- Printing a functor,
- Determining arity,
- Finding a procedure.

In particular, the representation of a call contains a pointer to an atom as the only handle on its procedure. Addition and deletion of clauses in the procedure does not therefore require modification of its calls.

Hashing is used to locate appropriate atoms during conversion from external representation. Such conversion takes place when terms are read in or when they are created by *functor* and *pname*. For simplicity, linear rehash is used in the current version: you might wish to improve it.

Print names are represented in CT by contiguous sequences of characters terminated with EOS characters (zero bytes). As a name is created by *pname* or the **READER**, its characters are pushed on top of the string area in CT (procedure *buildname*). On termination of the string, *wrapname* is invoked to locate an atom with the same printname. If such an atom is found, the string is obliterated; otherwise a new atom is created and returned. Since this atom's arity is unknown, the arity field is set to the special value of *noarity* (procedure *findname*).



Atoms are located by the READER in a two-phase process. First, *buildname* and *wrapname* are used to find the first atom with this name; then a single scan through the (virtual) hash chain finds an atom with the correct arity, or detects its absence and creates it (procedure *findatom*). Conversion between atoms of different priorities, needed to implement *functor*, requires invocation of the hash algorithm to locate the beginning of the appropriate hash chain (procedure *samename*).

Procedure descriptors are allocated in the Pascal heap. Descriptors are formed of lists of records, each of them with:

- a pointer to the next element in the list;
- the number of variables in an activation record;
- either the number of a system procedure, or pointers to the prototypes of a clause's head and body.

A system procedure descriptor is formed of a single such record. The descriptor of a Prolog procedure is a list of records, one for each clause. The head predicate's atom always points at the first element of this list.

A clause body is represented by the prototype of a Prolog list containing its calls. Figure 7.2a illustrates the layout (recall that the binary dot is the Prolog list constructor).

### 7.3.3. Prototypes and Term Instances

The main table, MT, holds a variety of objects which are distinguished partly by their addresses and partly by their contents. Addresses are used to distinguish between prototypes and term instances (fields denoting variables contain variable offsets in prototypes, and variable bindings in term instances). Prototypes of ground terms, which contain no variables, are also used as instances: this helps keep down the size of the copy stack.

Instances of non-ground terms are kept in the stack area. It is divided into the copy stack and the variable stack. The variable stack holds activation-record variables and is separated from the copy stack because it can shrink on procedure return and not only upon backtracking (see Chapter 6).

Object contents are used to distinguish between integers, variables, and "normal" terms with functors.

- Integers are two-word objects. The second word holds the integer and the first—a special marker INT, which prevents the interpreter from treating integers as pointers.

—Variables hold values less or equal to VARLIM (both INT and pointers to MT or AT objects have values greater than VARLIM). VARLIM is kept only inside the dummy variable (—) prototype, whose address is DUMVARX—this prototype is treated as ground. The value FREEVAR (equal to VARLIM - 1) fills free variable instances. Values below FREEVAR are negative: in prototypes their absolute values denote offsets in variable frames, and in instances their absolute values

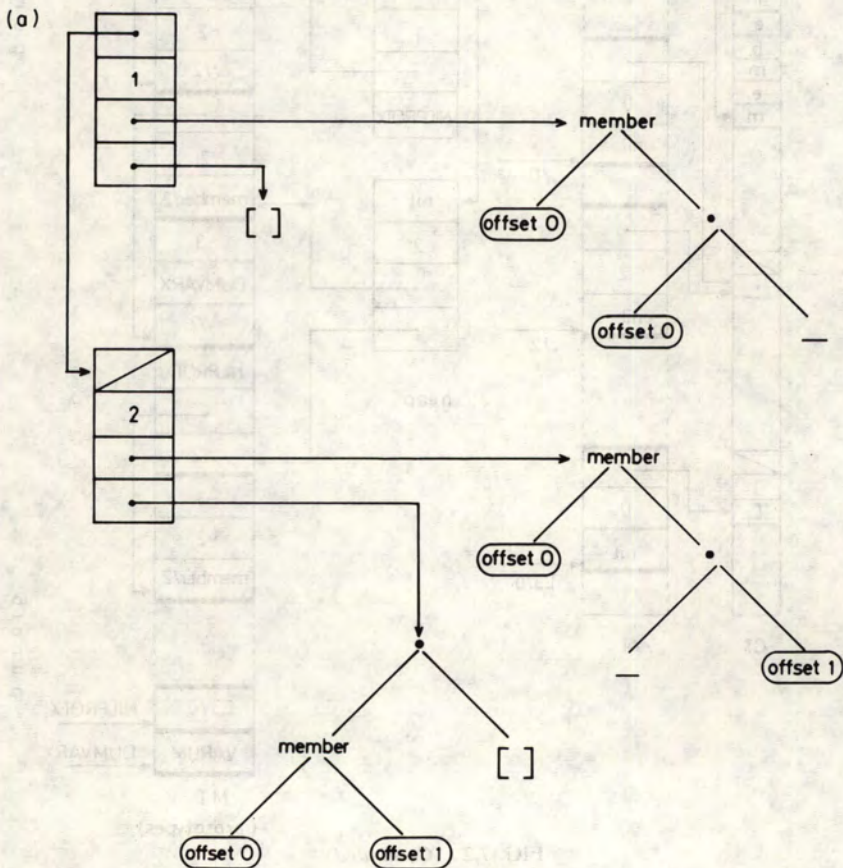


FIG. 7.2 The internal representation of *member*:

`member( :0, :0_ ) : [ ]`

`member( :0, _ :1 ) : member( :0, :1 ) . [ ]`

(a) The abstract form. (b) The data structures (variable offsets adjusted by `offset`; `[]/0`, `./2`, `./0`, `member/2` denote addresses). (continued)



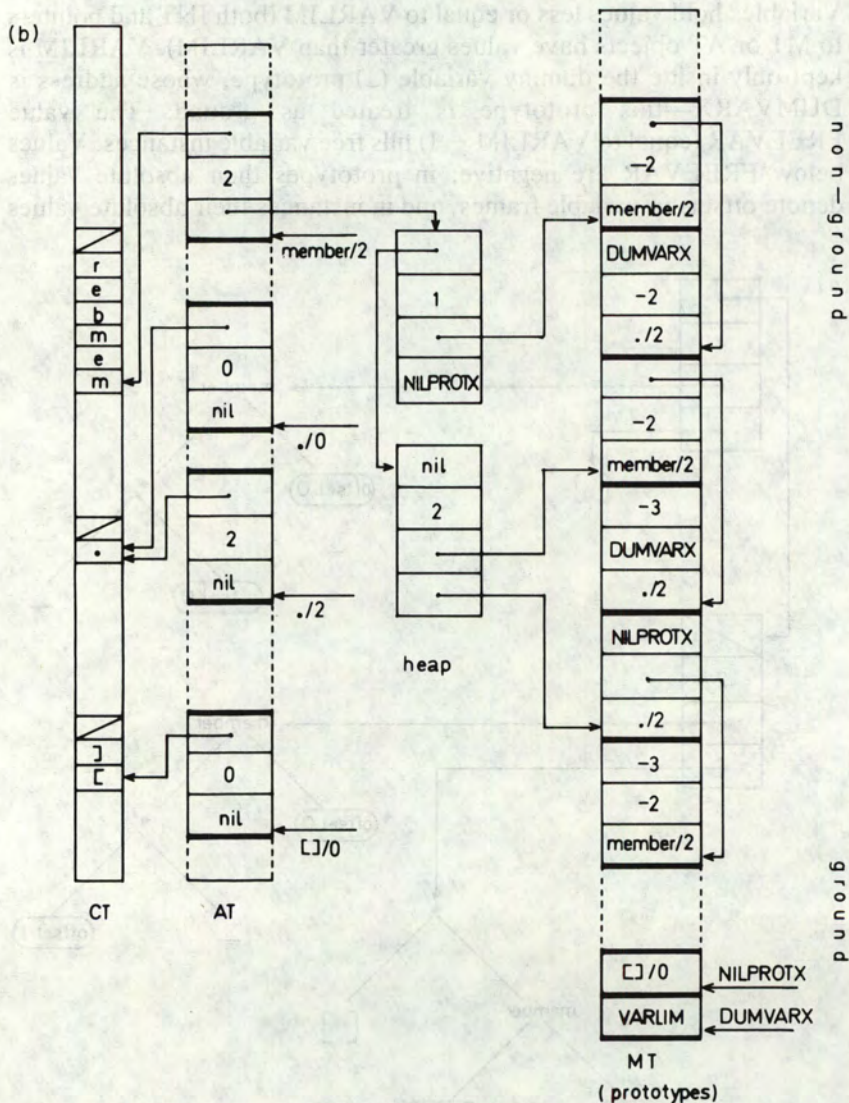


FIG. 7.2 (Continued)

are pointers to variable bindings. (Actually, the situation is slightly different:  $\text{INT} = 1$ ,  $\text{VARLIM} = 0$  and  $\text{FREEVAR} = -1$ . All negative entries denote non-dummy variables. MT's lower index is 2, but variable frame offsets start from 0 and are therefore adjusted by the constant  $\text{OFFOFF} = 2$ ;  $-2$  stands for offset 0,  $-3$  for offset 1, etc.)

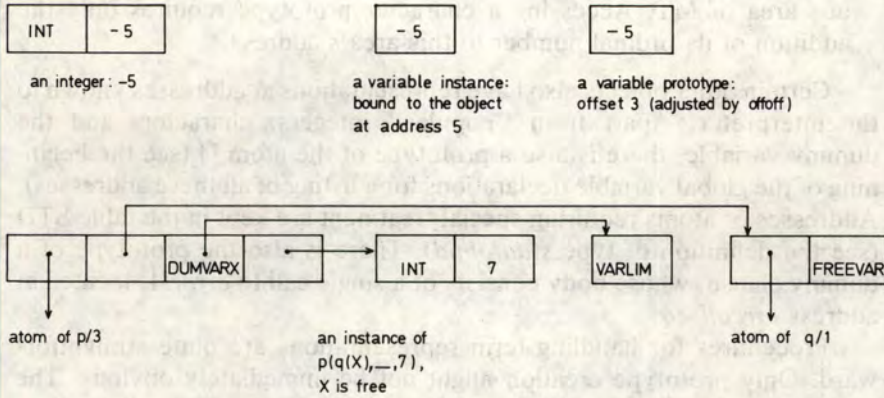


FIG. 7.3 Internal representation of terms.

—“Normal” terms are contiguous sequences of words. The first word holds a pointer to the main functor’s atom (its arity field defines the length of the sequence). Other words represent arguments. For variable arguments see above; other arguments are represented by pointers to appropriate objects (see procedures *getarity* and *getarg* in the listing).

Figure 7.3 illustrates these conventions. Figure 7.2b shows the complete internal representation of a procedure.

As explained in Chapter 6, term instances are pushed onto the copy stack only when absolutely necessary (when they become variable bindings) and are otherwise represented as in Structure Sharing. It is therefore convenient to represent all instances by a pair of pointers. If the first pointer addresses a prototype, the second (which we shall call the prototype’s **environment**) is a pointer to an area in the variable stack. If the first pointer addresses a term instance, the second is disregarded. Note that—unlike in Structure Sharing implementations—the environment need never change as term arguments are accessed: variable bindings are never nonground prototypes and require no environment.

The normal mechanisms of object recognition and creation are circumvented in two major cases (see procedure *loadsyskernel*).

- To avoid creation in the copy stack of too many integer objects representing intermediate results, a range of the most frequently used integers (−1..10 in this version) is maintained in the form of unique ground prototypes.
- To avoid the overhead of locating character atoms, checking whether functors represent characters, and duplicating character prototypes or instances, ground prototypes of ASCII characters are kept in a contigu-



ous area of MT. Accessing a character prototype requires only the addition of its ordinal number to this area's address.

Certain other objects also have representations at addresses known to the interpreter. Apart from "popular" integers, characters and the dummy variable, there is also a prototype of the atom [] (see the beginning of the global variable declarations for a listing of all these addresses). Addresses of atoms requiring special treatment are kept in the table STD (see the definition of type *stdatomid*). There is also the prototype of a dummy clause, whose body consists of a single call to *error/1*, located at address *errcallseq*.

Procedures for handling term representations are quite straightforward. Only prototype creation might not be immediately obvious. The method is quite similar to that used for creating entries in the dictionary. A prototype is allocated by invoking *initprot* with information about the main functor. Arguments are then filled in by *newparg* and *newpvararg*, and the process is terminated by *wrapprot*. This procedure checks if all the arguments are ground—when this is the case, the prototype is moved to the ground prototype area. Note that the process is inherently recursive, as argument prototypes may be created before their parent term's prototype is wrapped up. This is why *initprot* must be used: piecemeal allocation would not preserve contiguity.

A short comment about *terminst*, the procedure usually used to create terms on the copy stack. Non-variable arguments are represented not by direct pointers, but by negative values, as if they were all formed by instantiating pre-existing variables. This is necessary because the procedure *argument* (which follows chains of variable bindings to locate the final instantiation) expects variable arguments directly inside the representation of their parent terms. A recursive call on *terminst* can return a variable and treating the variable as a normal argument—by inserting a positive pointer to it—would break the chain of references. (Such things are not easily seen, and the erroneous situations are rather infrequent: this bug was the hardest to locate!)

### 7.3.4. Control

In Toy, clause bodies are represented as prototypes of lists. The list elements are prototypes of calls, and none of them is an integer or a variable. While not directly related to the external form of clauses in Prolog-10, this representation is very regular and easy to handle.

The method of representing control state is almost exactly like that described in Chapter 6. The principal difference is that the variable part of



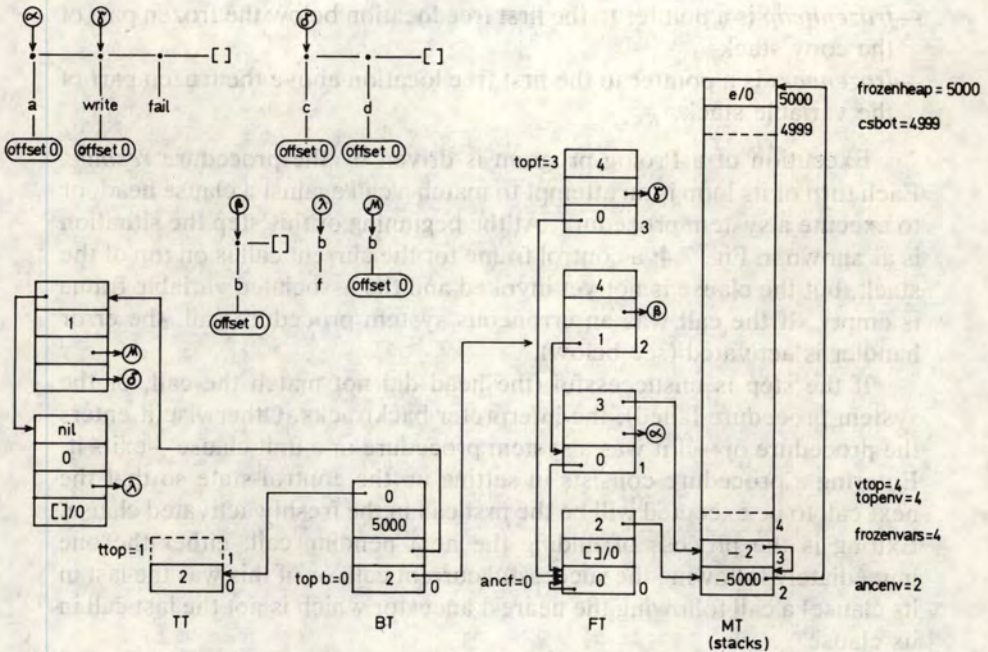


FIG. 7.4 A more detailed form of Fig. 6.6d.

an activation record is kept on a separate stack. Figure 7.4 is a detailed version of Fig. 6.6d. We shall comment only on the variables used as “control registers”.

The crucial variables are:

- topf*, a pointer to the current control frame (i.e. activation record), which is always on top of the stack in FT;
- topb*, a pointer to the current backtrack point (i.e. fail point) record, which is always on top of the stack in BT;
- csbot*, a pointer to the first free location below the copy stack in MT (this stack grows downwards);
- vtop*, a pointer to the first free location above the variable stack in MT;
- ttop*, a pointer to the first free location above the trail stack in TT.

Five auxiliary variables contain copies of information available elsewhere. They are used for efficiency:

- ancf* is a pointer to the current control frame’s parent frame;
- topenv* is a pointer to the current variable frame (associated with *topf*);
- ancenv* is a pointer to the variable frame associated with *ancf*;



- frozenheap* is a pointer to the first free location below the frozen part of the copy stack;
- frozenvars* is a pointer to the first free location above the frozen part of the variable stack.

Execution of a Prolog program is driven by the procedure *resolve*. Each turn of its loop is an attempt to match a call against a clause head, or to execute a system procedure. At the beginning of this step the situation is as shown in Fig. 7.4: a control frame for the current call is on top of the stack, but the clause is not yet invoked and the associated variable frame is empty. If the call was an erroneous system procedure call, the error handler is activated (see below).

If the step is unsuccessful (the head did not match the call, or the system procedure failed), the interpreter backtracks. Otherwise it enters the procedure or—if it was a system procedure or a unit clause—exits it. Entering a procedure consists in setting up the control state so that the next call to be executed will be the first call in the freshly activated clause. Exiting is the process of finding the next pending call: either the one immediately following the successful current call, or (if this was the last in its clause) a call following the nearest ancestor which is not the last call in its clause.

To stop the execution, the flag *stop* must be set. This is done either by the system procedure *halt*, or by *backtrack* when there are no fail points left (i.e. when the directive failed) or by *exit* when it cannot find a pending call (i.e. when the directive succeeded).

Two auxiliary variables play the role of a program counter:

- ccall* contains a pointer to the prototype of the current call (it is the prototype of the first element in the list indicated by the current control frame's *calls* field, unless that element is an invocation of *call* or *tag*: *ccall* is then the outermost argument which is neither of these);
- cproc* contains a pointer to the descriptor of the procedure invoked by *ccall* (for Prolog procedures, this is the first clause's descriptor when in forward execution, and a pointer recovered from a backtrack point record's *resume* field when immediately after a failure).

Notice that a fail point's *resume* field points at the predecessor of the clause which is to be retried. This is so to make *retract* correct.

The algorithm used for tail recursion optimisation (procedure *trooverlay*) merits some explanation. We employ the naive method suggested by Fig. 6.7. After unification is over, procedure *candotro* checks whether the current call is an untagged tail call and whether the ancestor frame is not frozen. If so, neither the call nor the variable frame associated with the ancestor frame will ever be needed again. The current



variable frame is shifted to replace the ancestor variable frame, and the control stack is popped so that the ancestor control frame becomes top-most (the most recently activated clause is still accessible through *cproc*). The algorithm is made a little complicated by the fact that the shifted variables may be instantiated to one another or to the destroyed (overlaid) variables. Both cases are illustrated in Fig. 7.5.

The cut procedure simply removes as many backtrack point records as necessary (possibly none) to ensure that the call invoking the procedure containing the cut—and all subsequent calls—will not be retried. (There are exceptions to this rule: notice that *,/2*, *;/2* and *call/1* are transparent to the cut.) After popping off backtrack points, the interpreter must purge the topmost section of the trail to remove references to variables which are no longer frozen. This is necessary, because such variables can be popped off, or shifted, during TRO. Notice that the method of TRO applied

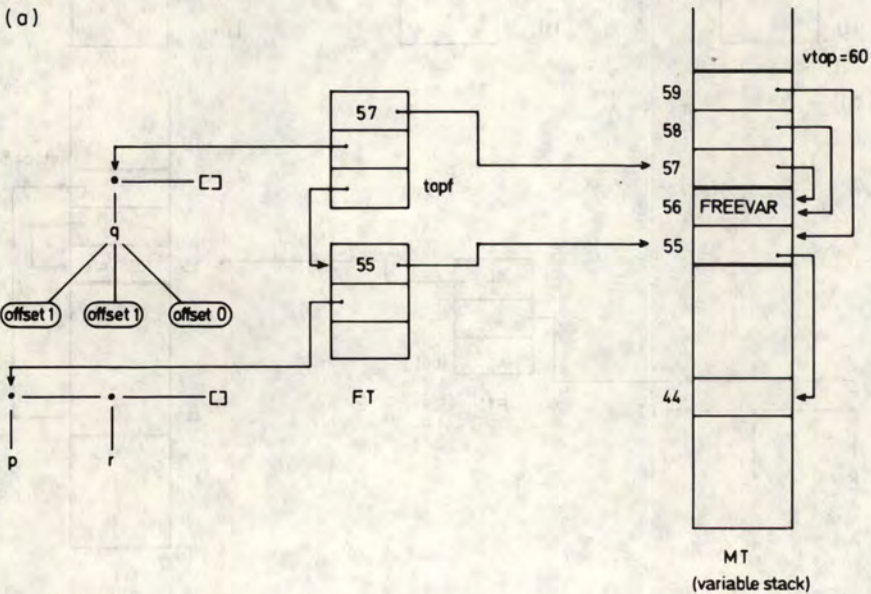


FIG. 7.5 Tail recursion optimisation: merging two frames. (a) The initial situation. Both frames are not frozen, the call is tail recursive. The variables at 57 and 58 are instantiated to the same free variable, the variable at 59 is instantiated to the variable at 44. (b) Adjustment pass. (i) The first variable (at 57) points at an overlaid free variable (at 56). The direction of the pointer is reversed. (ii) The second variable (at 58) is dereferenced to that at 57 through that at 56. The reference is remapped: the second variable points at 55—the *future* location of the variable now at 57. (iii) The third variable (at 59) is dereferenced to that at 44 through that at 55. (c) Shifting pass overlays the parent's variable frame with the current variable frame; the parent's control frame becomes current. (*continued*)



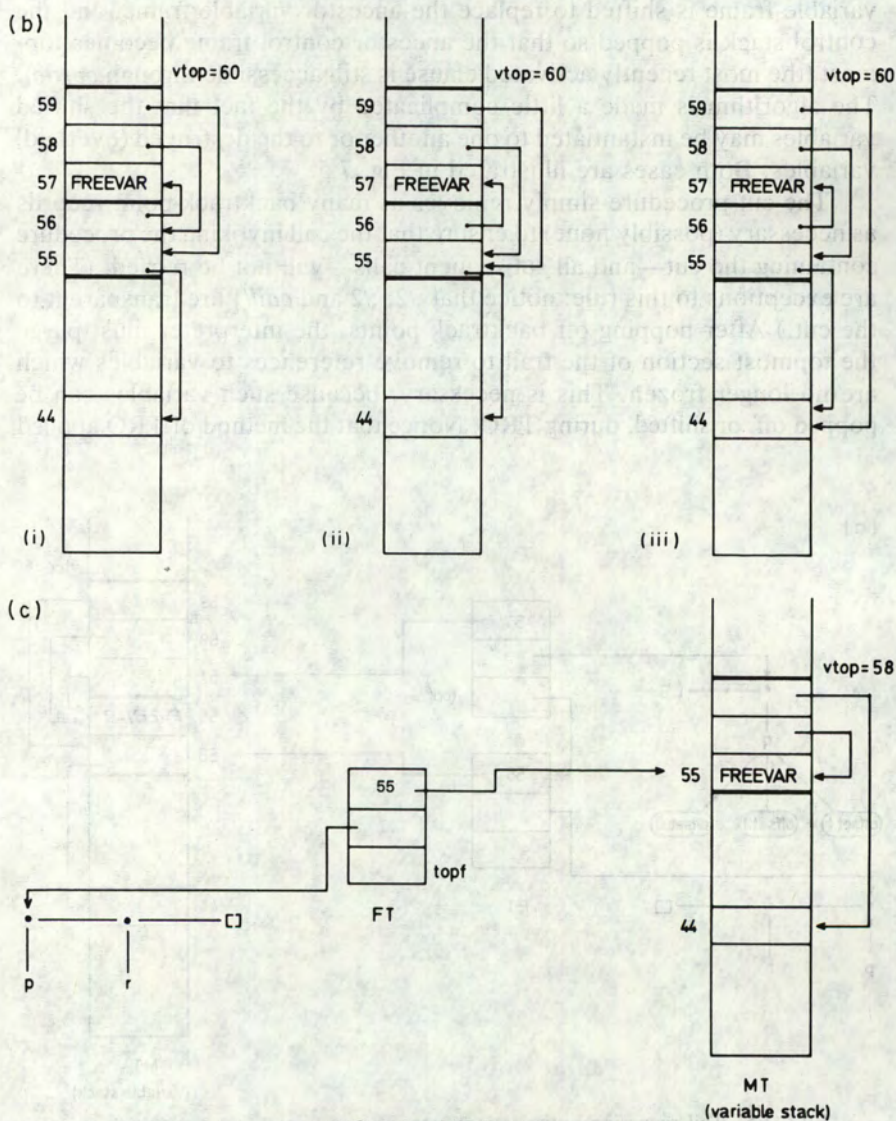


FIG. 7.5 (Continued)

here makes it fairly easy to perform **delayed** frame merging after things are made deterministic by the cut. We shall not enter into the details of this and of *tagcut*: this is a simple exercise.

The last thing worth mentioning is the handling of erroneous calls to

system procedures. This involves pushing a dummy variable frame, with a single variable instantiated to the erroneous call. The current control frame (in which the call was invoked) is associated with this variable frame and becomes the ancestor of *error/1*. As a result, the parameter of *error/1* is the right instance of the erroneous call. The process is illustrated in Fig. 7.6.

The program maintains several important invariants, such as “there are no outside references to non-frozen variable frames except from variables higher in the variable stack”. We decided to let you have the fun of discovering them for yourself (after all, these are the real trade secrets).

### 7.3.5. System Procedures

We shall not give a detailed description of the system routines. There are too many of them, and the listing is more or less self-explanatory. The general principles are as follows:

- All system procedures are invoked through procedure *sysroutcall*;
- sysroutcall* sets up pointers to their parameters in table SPAR (the values of integer parameters are also passed through table SPARV);
- System procedures that can fail or succeed indicate the result by setting a Boolean parameter (*success*) passed by *sysroutcall*;
- Whenever a system procedure detects an error, it sets the global flag *syserror*, which forces the interpreter to invoke *error/1* (see the end of the previous section).

There are no tricks, except in the procedure concerned with creating new clauses. It is important that several occurrences of a variable be represented by occurrences of the same offset when a term is translated into a prototype. To achieve this, addresses of variables appearing in an asserted clause are stacked in the free area above the topmost variable frame. With each variable occurrence, this temporary variable dictionary is searched linearly and, possibly, augmented. The position of a variable in this dictionary is treated as its offset.

To add a new system procedure, one must:

- Write its code;
- Insert its identifier in type *sysroutid* (its place there defines its position);
- Insert its call in procedure *sysroutcall* (in the same position);
- Insert its name and arity in the kernel file (in the same position)—see the next section.



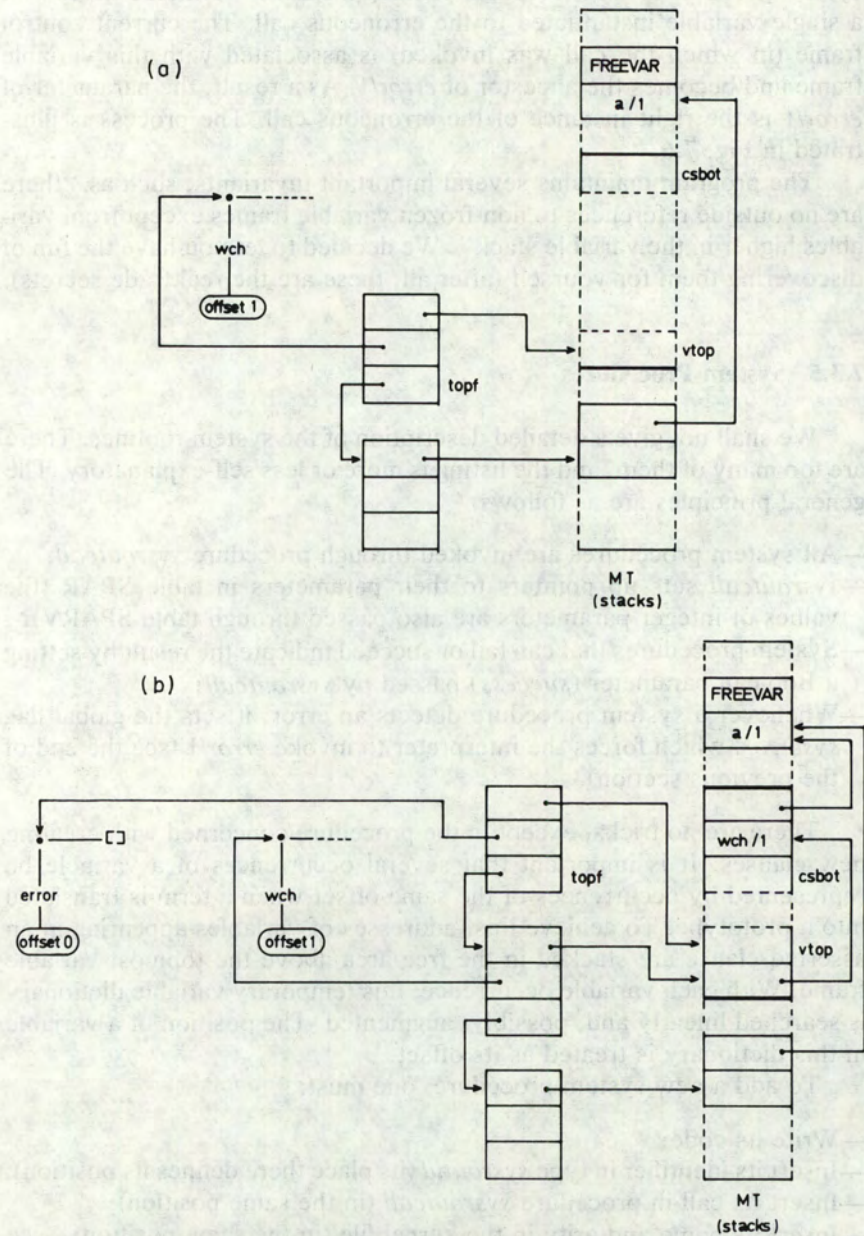


FIG. 7.6 Handling erroneous calls to system routines. (a) *wch* detects an incorrect argument: *a(V)*. (b) a call to *error/1* is set up.

### 7.3.6. Initialisation

Initialisation is done in three phases. First, most of the variables are set by procedure *initvars*. Then two portions of data are read from the so-called "kernel file". One portion defines the names and arities of standard atoms whose addresses must be known to the interpreter. They are created and their addresses are stored in table STD. The other portion defines the names and arities of system procedures: as the atoms are created, they are associated with system procedure descriptors. The number and order of all these atoms is known to the interpreter. Arities are important, but printnames are arbitrary and *can be changed at will*.

The last phase of initialisation consists in creating a number of standard objects. Their addresses are known to the interpreter but they cannot be created before the addresses of standard atoms are fixed. The objects are:

- The prototype of [];
- The prototypes of characters;
- The prototypes of the integers  $-1, 0, \dots, 10$ ;
- The dummy clause body used to invoke *error* (it is the prototype of [error(X)]);
- The prototype of *user*, needed by the stream switching procedures (see Section 5.7.1).

After initialisation, the interpreter begins normal execution, reading the current file. This is normally the kernel file, containing some useful library procedures. One can also append the bootstrapper or the translated monitor (see below).

## 7.4. INTERPRETATION OF PROLOG-10 IN TOY-PROLOG

### 7.4.1. Intermediate Language

Even a modest program in Toy-Prolog can be unmanageable. To write the monitor, we use a subset of full Prolog, without operators and grammar rule notation. Commas and the symbol  $:-$  are treated as separators. List notation is allowed, with one restriction: an X in [... | X] must be a variable. This subset is translated into Toy-Prolog by a "bootstrapper" written in Toy-Prolog. Debugging and testing the monitor required frequent retranslations of its small pieces, but the gain in readability was worth



this extra effort. Of course, once the monitor works, the bootstrapper is no longer needed.

The bootstrapper is listed in Appendix A.2. Comments starting with *%%* associate mnemonics with variable numbers. The main procedure is *translate* (lines 2–13), with two parameters—the names of the source and output files. The unit processed with each turn of the failure-driven loop is a single clause or a comment. The loop stops upon encountering a *@* in place of the first non-blank character of a unit. The translation of a clause is a string which is built “on the fly” on a difference list of characters; the list is represented by the two parameters christened *termrepr* and *rest\_of\_termrepr*. Here is how the clause in lines 54–55 would look after rewriting it into full Prolog and combining those parameters:

```
ctailaux( Fterm_firstch, Termrepr -- Rest_of_termrepr,
          Sym_tab ) :-
    fterm( Fterm_firstch, Fterms_firstch,
          Termrepr -- [ ' ', '.' | Middletermrepr ], Sym_tab ),
    fterms( Fterms_firstch, Middletermrepr -- Rest_of_termrepr,
           Sym_tab ).
```

*Fterms\_firstch* is the first non-blank following a functor-term; in a correct clause, it can only be a dot, or a comma (see lines 58–66).

Comments embedded in a clause are copied at once (lines 50–53). Moreover, the string contains end-of-line and blank characters which improve the appearance of the translation.

Error in a clause causes a message to be printed and the input up to the nearest dot to be reprinted and skipped (see lines 15–21). The program assumes the data are correct, and protests upon encountering the first unexpected character.

Output for each clause with variables is followed by a comment that associates variable numbers with source names taken from a symbol table for this clause (lines 219–226). The table is an open list of names. Their positions are used as variable numbers in the translation. Up to 99 variables can occur in a clause. The number–name pairs are written six in a line (line 224).

There are some other minor points worth noticing. For example, the output string gets closed eventually due to the *[]* in the initial call on *clause* (line 11); translations of lists within lists are parenthesized, see the fifth parameter of *term* (lines 131, 136, 137); identifiers are enclosed in quotes by *fterm* (lines 69–70); etc. etc. However, the rest of the program should be self-explanatory. A hint: it can be viewed as a metamorphosis grammar used for synthesis, driven by input data, with the two compo-



nents of a difference list serving as an input and output parameter (see Section 3.1).

### 7.4.2. Overview of the Monitor

The core of the monitor is an implementation of the built-in procedure *read* that is used in user programs (see Section 7.4.3). The user communicates with Prolog via an interactive “driver” which operates in a loop terminated by executing the procedure *stop*. In each cycle the driver prompts the user with

?-

and then reads and executes a directive. The symbol table (returned by the two-parameter *read*; see the end of the next section) pairs source names of variables with variables proper. After successful execution, the symbol table is used to display final instances of these variables, and the driver awaits a printable character. If it is a semicolon, execution resumes with forced failure, else processing of this directive terminates.

A directive can be prefixed with *:-* (we call such a directive a *command*, and that without the prefix a *query*). It will then be executed deterministically, and variable instances will not be printed. However, neither a non-unit clause nor a grammar rule make sense when read directly by the driver: a two-parameter procedure *:-* or *-->* (presumably undefined) would be called. User procedures can be defined by calling the built-in procedure *consult* or *reconsult*; both are implemented in the driver. In “consult mode”, term *L --> R* is treated as a grammar rule and translated by the procedure *transl\_rule* (see Section 7.4.4). A one-parameter term *:-C* is treated as a command and executed. Other terms are treated as program clauses.

The monitor is listed in Appendix A.3.

### 7.4.3. Reader

The syntax of Prolog-10 is only deceptively simple, so the reader is rather involved. One wonders whether a simpler syntax would necessarily be less user-friendly.

The main component of the reader is a parser which produces internal representations of terms on input (Appendix A.3, lines 90–332). Translation of an internal representation into a term proper is quite straightforward (look at the listing of the procedure *maketerm*, lines 334–357, after reaching the end of this section).



The parser is a classic operator precedence parser; those parsers belong to the “shift–reduce” class—they are bottom-up and deterministic (Gries 1971, Aho and Ullman 1977).

Recall that, roughly, an operator precedence grammar has no production with two consecutive non-terminals, and all its productions are such that a shift–reduce parser can determine the handle by comparing neighbouring terminals in a sentential form. This is possible when each pair of terminals is in at most one of the three relations denoted by  $<$ ,  $=$ ,  $>$ . The relations are defined as follows ( $p, q$  are terminals,  $U, V, W$  non-terminals):

— $p = q$  if there exists a production of the form

$$\begin{array}{l} U \rightarrow \dots pq \dots \\ \text{or } U \rightarrow \dots p V q \dots \end{array}$$

— $p < q$  if there exists a production of the form

$$\begin{array}{l} U \rightarrow \dots p V \dots \\ \text{where } q \dots \text{ or } W q \dots \text{ can be derived from } V \end{array}$$

— $p > q$  if there exists a production of the form

$$\begin{array}{l} U \rightarrow \dots V q \dots \\ \text{where } \dots p \text{ or } \dots p W \text{ can be derived from } V \end{array}$$

A parser shifts (i.e. scans a sentential form from left to right) until it detects a pair of terminals related by  $>$ . It then scans backwards until the nearest pair of terminals related by  $<$ . The  $<$  and  $>$  are assumed to be brackets delimiting the handle in a canonic parse: the handle is reduced and the process continues.

Note that  $<$ ,  $=$  and  $>$  have nothing to do with the common number-ordering relations. However, if terminals are operators as in arithmetic expressions, these relations reflect operator priority: the grammar is structured so that higher priority operators (with operands) are reduced first. The situation is similar in the case of Prolog “operators” (even though in Prolog-10 weaker operators are given the higher priority). We shall say—very informally—that  $f$  is weaker than  $g$  if  $f < g$  or  $g > f$ . But note that, for example,  $+ < ($ ,  $( = )$ , and  $+ > )$ .

We shall now return to our program. We assume that the input is delimited by two additional operators. The rightmost delimiter is weaker than any operator to its left; the leftmost is weaker than any operator to its right (except the other delimiter). Notice that an empty input is erroneous.

The parser maintains a stack of symbols. Initially the stack contains



only the leftmost delimiter. The first true terminal becomes the current input terminal. In each step, the current input terminal is compared to the topmost terminal on the stack. Three situations are possible:

1. The input is erroneous—the parser stops “with error”;
2. The topmost terminal is stronger—there must be a production with the righthand side consisting of a number of topmost symbols on the stack; we reduce the stack by replacing all these symbols with a corresponding lefthand side;
3. The topmost terminal is not stronger, i.e. no righthand side has been completed—we shift the current input terminal onto the stack and make the next terminal current.

Our operator grammar of Prolog-10 terms assumes seven classes of terminals and one class of nonterminals, *t* (for terms). Parameters of symbols are used to build the internal representation of a given term.

Terminal symbols are read by a scanner (see Appendix A.3, lines 361–480). The procedure *absorbtoken* (lines 379–409) reads and constructs a “raw” token:

—id(NameString)	from words, symbols, and solo-characters;
—qid(NameString)	from quoted names;
—var(NameString)	from variables;
—num(NumberString)	from integers;
—str(String)	from strings;
—br(LeftRight, Type)	from brackets (LeftRight is l or r, Type is '()', [], or '{}');
—bar	from  ;
—dot	from a full stop.

Next, the procedure *maketoken* (lines 457–480) constructs a terminal symbol:

—vns(Variable)	from var(NameString);
—vns(Number)	from num(NumberString);
—vns(String)	from str(String);
—ff(Name, Types, Priority)	from id(NameString) (when this functor is an operator);
—id(Name)	from id(NameString) (when this functor is not an operator) and from qid(Name- String) (i.e. a quoted name never de- notes an operator);
—br(LR, T)	from br(LR, T);



—bar from bar;  
 —dot from dot.

The terminal symbol *dot* is used as the rightmost delimiter of the input. The leftmost delimiter (and the seventh terminal) is *bottom*. It is never returned by the scanner: the parser's main procedure, *gettr*, pushes it onto the initially empty stack. Both delimiters never appear in productions.

The Types argument of *ff* is a list of functor types: [Binary], or [Unary], or [Binary, Unary] (see the definition of the built-in procedure *op* lines 656–718)).

A symbol table in an open list is used to relate a variable's name to a Prolog variable.

The grammar underlying the parser is given in the listing (lines 99–107). The definition of internal representation can be read off the *reduce* procedure (lines 158–179). Incidentally, the procedure can be paraphrased as a metamorphosis grammar. For example, the fifth and sixth clause would be rewritten as

$$\begin{aligned} t(\text{tr}(\text{Type}, X)) &\rightarrow [\text{br}(1, \text{Type})], t(X), \\ &\quad [\text{br}(r, \text{Type})]. \\ t(\text{bar}(X, Y)) &\rightarrow [\text{br}(1, [])], t(X), [\text{bar}], \\ &\quad t(Y), [\text{br}(r, [])]. \end{aligned}$$

Notice, however, that top-down analysis based on such a grammar would not be deterministic.

There are five types of internal representations:

- arg0(X) for X a variable, name, string, or nullary functor;
- tr1(Name, X) for a prefix or postfix term (X is the representation of the argument);
- tr2(Name, X, Y) for an infix term (X, Y are the representations of the arguments; in particular, the comma is an infix functor, so “comma-lists” of terms are represented with tr2—for example, the representation of
 

a, b, c  
 is  
 $\text{tr2}(\text{' , '}, \text{arg0}(a), \text{tr2}(\text{' , '}, \text{arg0}(b), \text{arg0}(c)))$
- bar(X, Y) for a list with front X and tail Y; X is often the representation of a comma-list;

—`tr(Name, X)` for all other valid situations:  
     `tr('()', X)` is equivalent to `X`;  
     `tr([], X)` represents a list (of definite length), `X` usually represents a comma-term  
     `tr('{}', X)` represents the term `{(Cond)}` where `Cond` is the term represented by `X` (this is used in grammar rules);  
     `tr(Name, X)` with `Name` other than a bracket type (and `X`—usually the representation of a comma-term) represents a normal term; for example, the term

`foo( 'p', 5)`

is represented by

`tr( foo, tr2( ', ', arg0( [ p ] ), arg0( 5 ) ) )`

The parser's entry point is the procedure *gettr* (lines 125–127), and the main loop is implemented as the procedure *parse* (lines 129–138). The loop terminates successfully when the original input (*bottom* and *dot* included) reduces to the sequence

`bottom    t( InternalRepresentation )    dot`

The parser fails in two situations:

- when the procedure *establish\_precedence* fails, i.e. when the topmost terminal on the stack and the current input terminal do not compare;
- when the procedure *reduce* fails, i.e. the top segment of the stack does not match any production.

The procedure *topterminal* (lines 140–143) returns `Top`, the topmost stack terminal, and its position: 1 means `Top` is the top item, 2 means it is covered by a `t(_)`.

The precedence relations are summarized in Table 7.1. We treat all operators jointly with respect to other terminals. Empty slots signify erroneous combinations of contiguous terminals.

A functor–functor relationship is the only potentially conflicting one: to establish the precedence relation for a given `Top` and `Input`, we must consider their priorities and types (sometimes even some broader context should be considered but this might require changes in the otherwise deterministic algorithm). If the priorities differ, the functor with lower priority is taken as stronger, according to the conventions of Prolog-10. (Notice, however, that when `Top` is stronger, `Input` cannot be a



TABLE 7.1

Precedence Relations for the Operator Grammar of Terms

Top	Input												
	vns	id	(	)	[		]	{	}	ff	bottom	dot	
	vns				>		>	>		>	> <sup>b</sup>		>
	id			=	>		>	>		>	> <sup>b</sup>		>
	(	<	<	<	=	<			<		<		>
	)				>		>	>		>	> <sup>b</sup>		>
	[	<	<	<		<	=	=	<		<		>
		<	<	<		<		=	<		<		>
	]				>		>	>		>	> <sup>b</sup>		>
	{	<	<	<		<			<	=	<		>
	}				>		>	>		>	> <sup>b</sup>		>
	ff	< <sup>a</sup>	< <sup>a</sup>	< <sup>a</sup>		< <sup>a</sup>			< <sup>a</sup>		< <sup>a</sup> > <sup>b</sup>		>
	bottom	<	<	<	<	<	<	<	<	<	<	<	>
	dot												

<sup>a</sup> Top can be any prefix or infix functor, i.e. Types = [xf] and Types = [yf] are excluded.

<sup>b</sup> Input can be any infix or postfix functor, i.e. Types = [fx] and Types = [fy] are excluded.

prefix functor, and when Input is stronger, Top cannot be postfix.) If Top and Input have equal priorities, their types must be examined (see below).

Mixed<sup>1</sup> functors require special treatment. In most contexts, their inherent ambiguity is apparent: only one of a functor's types can be properly attributed to it. For example, let Input be &, an [xfy, fy] functor, and Top a left parenthesis not covered by a non-terminal:

..... ( & .....

Surely, & can only be a prefix variation of this mixed functor—an infix variation is excluded. Likewise, if Top is \$, an [xfx, xf] functor, covered

<sup>1</sup> Recall that our version of Prolog allows a mixed functor to have only one binary and one unary type, both with the same priority.

by a non-terminal, and Input a right bracket:

..... \$ Term ] .....

then \$ certainly cannot be a postfix functor. In such situations, we can “disambiguate” the mixed functor by removing the incompatible type from its representation. For example, we replace  $\text{ff}(\&, [\text{xfy}, \text{fy}], \text{Priority})$  with  $\text{ff}(\&, [\text{fy}], \text{Priority})$ .

The relation in Table 7.1 is implemented by the procedure *establish\_precedence* (lines 195–204), which takes the two terminals and the position of Top. It fails given an incorrect combination, otherwise it succeeds with the fourth parameter instantiated as *gt* (Top is stronger) or *lseq* (Top is not stronger). When both terminals are mixed functors, the procedure tries to disambiguate their types. The last two parameters are instantiated as the new top and new input terminal, to be used in the next step (usually they remain unchanged).

The real job is done by the procedure *p* which returns *gt* or *lseq*, or—when functors are involved—*gt*(NewTop, NewInput) or *lseq*(NewTop, NewInput). It fails given an erroneous pair of terminals.

Table 7.1 has 80-odd nonempty entries, but it can be easily simplified. First of all, we can treat *bottom* and *dot* separately; see the last two clauses of *p* (lines 240–241). Next, we consider slots with “=”—the first four clauses (lines 206–209) take care of this, and the remainder of *p* can operate with the six slots cleared. Now we are left with a  $10 \times 10$  table with three different rows and three columns. Table 7.2 depicts the situation after combining identical rows and columns.

**TABLE 7.2**  
**Simplified Precedence Relations**

Top \ Input	vns id ( [ { ) ] }	ff
vns id ) ] }		$>^a$
( [ {	$<^a$	$<$
ff	$<^c$	$>$

<sup>a</sup> Top and Input cannot be separated by a non-terminal.

<sup>b</sup> Input cannot be a prefix functor.

<sup>c</sup> Top cannot be a postfix functor.



The next six clauses of  $p$  (lines 211–222) take care of the six nonconflicting slots in Table 7.2. The procedure *restrict* (lines 265–271) is used to test and possibly disambiguate the type of a functor. The procedure performs set subtraction for sets given as lists; it will fail if the difference is an empty set.

Now we must try to resolve a conflict in the remaining slot. A closer look at the grammar allows a refinement of this slot (see Table 7.3). The 12th and 13th clauses of  $p$  (lines 229–238) are responsible for situations when the priorities differ. Again, we also attempt a disambiguation of types.

The 11th clause (lines 225–227) applies to functors with equal priorities. Table 7.4 shows the precedence relation in this case. We allow all combinations that can be disambiguated without analysing broader context to the left or to the right of the two functors. For example, an xfy functor  $f$  is weaker than an xfx functor  $g$  because the term

$A\ f\ B\ g\ C$

cannot be interpreted as

$(A\ f\ B)\ g\ C$

— $g$ 's left argument would have, incorrectly, the same priority as  $g$ .

The relation of Table 7.4 is implemented by the procedure *ff\_p* (lines 319–332), which returns *lseq*, *gt* or *err*. Conflict resolution is performed by the procedure *res\_confl* (lines 273–291), which also returns *lseq*, *gt* or *err* (*err* is later rejected by *do\_rels* called in  $p$ ). It also returns disambiguated—sometimes unchanged—functors.

If only one of the terminals is a mixed functor, we choose a non-conflicting interpretation by comparing slots in Table 7.4. This is done by

**TABLE 7.3**  
**A Refinement for Two Operators**

Input Top	prefix	infix	postfix
prefix	<	< >	< >
infix	<	< >	< >
postfix		>	>

TABLE 7.4

Precedence Relations for Operators with Equal Priorities

Top's type \ Input's type	xfy	xfx	xf	yfx	yf	fy	fx
yfx				$>^a$	$>^a$		
xfx				$>^a$	$>^a$		
fx				$>^a$	$>^a$		
xfy	$<^a$	$<^a$	$<^a$			$<^b$	$<^b$
fy	$<^a$	$<^a$	$<^a$			$<^b$	$<^b$
yf				$>^b$	$>^b$		
xf				$>^b$	$>^b$		

<sup>a</sup> Top and Input must be separated by a non-terminal.<sup>b</sup> Top and Input must not be separated by a non-terminal.

the procedure *match\_rels* (lines 297–300). For two mixed functors we extract a subtable of relations for each possible pair of interpretations; see Table 7.5 (and lines 286–289). The situation is clear if all four slots are the same. Otherwise there are only four patterns which can be correct: when one of the rows or one of the columns contains two *err* slots. Details—in the procedure *res\_mixed* (lines 302–317).

The procedure *read*(Term, SymbolTable) performs the two phases of the reader—see lines 65–69. It returns the symbol table with variables from this term. The table is used by the interactive driver (see Section

TABLE 7.5

The Subtable Template for Two Mixed Functors:  
the Binary and Unary Types Are Compared with  
Each Other

	TInpBin	TInpUn
TTopBin	RelBB	RelBU
TTopUn	RelUB	RelBB



7.4.2). If data are incorrect, the parser will stop on the first bad symbol and *read/2* will skip characters up to the nearest full stop after this symbol (which may also be a full stop). The built-in procedure *read/1* simply encapsulates *read/2*.

#### 7.4.4. Grammar Preprocessor

The grammar rule preprocessor (lines 482–583 in Appendix A.3) operates according to the principles presented in Chapter 3. The list of lefthand side terminals (usually empty) is connected to the output variable of the lefthand side non-terminal. Calls on the procedure *terminal* (Section 3.1) are “pre-executed” for efficiency. By way of explanation, here are two examples. The rule

$$a \rightarrow [p], b, [q, r], c.$$

is translated into

$$a([p|X], Z) :- b(X, [q, r|Y]), c(Y, Z).$$

The rule

$$a \rightarrow b, [q, r], c, [s].$$

is translated into

$$a(X, Z) :- b(X, [q, r|Y]), c(Y, [s|Z]).$$

(The translation of a list of terminals is *true*, absorbed by the next item’s translation; see *combine*, lines 540–542).

Conditions/actions (other than a single cut) are passed to the preprocessor as ‘{}’(C); see the procedure *maketerm* in the reader, lines 345–346). The functor ‘{}’ is stripped off by the procedure *transl\_item*, line 550.

Righthand sides separated by semicolons are preceded by a non-terminal defined as

$$'dummy' \rightarrow [].$$

This is necessary when alternatives start with different terminals. For example, the rules

$$a \rightarrow [p], b. \quad \text{and} \quad a \rightarrow [q], c.$$

would be translated with

$$a([p|Y], Z) \quad \text{and} \quad a([q|Y], Z)$$



as a lefthand side. Consequently the rule

$$a \rightarrow [p], b; [q], c.$$

must be translated as

$$\begin{aligned} a(X, Z) ; & \text{'dummy'}(X, [p | Y]), b(Y, Z) ; \\ & \text{'dummy'}(X, [q | V]), c(V, Z). \end{aligned}$$

For simplicity, this has been applied to all rules with alternatives.

#### 7.4.5. Library

The library (Appendix A.3, lines 585–1002) contains definitions of about 20 built-in procedures (note that several simple procedures are also defined in the kernel file, appendix A.1). Their definitions in Chapter 5 can be treated as design documentation. Their implementation is largely straightforward. We shall comment on a few not quite obvious passages.

The procedures `clause(Head, Body)` and `retract(Clause)` are “back-trackable”, i.e. can be used in failure-driven loops that generate or remove all matching clauses. Here is a description of the generator (the other procedure is programmed similarly). We are going to visit all clauses of a procedure and suspend execution each time we get to a matching one. This is achieved by setting up a recursive loop with its step distributed between two clauses (see the procedure `remcls/7`, lines 814–822). The first clause does the matching. Upon mismatch, we immediately proceed with the second clause, i.e. conclude the step. If the matching succeeds, the generator succeeds, too, but with a pending alternative. A failure later on resumes the second half of the step.

The procedures `write` and `writeln` both encapsulate the procedure `outterm(Term, With_or_without_quotes)` which first uses `numbervars` (lines 623–632) to bind all variables in `Term`, and next calls

```
outt( Term_after_numbervars, Context, With_or_without_quotes ).
```

`Context` specifies the essential features of a functor whose argument is `Term`. If it is not an operator, or there is no external functor, then `Context` is `fd(, )`. Otherwise, `Context` is `fd(ff(Priority, Associativity), Dir)`. `Term` may be to the left (`Dir = l`) or to the right (`Dir = r`) of the functor. Associativity may be `a(l)` or `a(r)` for left- and right-associative functors, and `na(l)`, `na(r)`, or `na(,)` for non-associative functors. `Context` is tested by the procedure `outff/5` (lines 933–935) to decide whether `Term` should be parenthesized to avoid ambiguity in the case of equal priorities. Actually, the



test—performed by *agree* (lines 939–943)—is rather crude (see the previous section!): sometimes we overparenthesize. The parameter of *na* has only been added for homogeneity, but it could be used in a more subtle detection of non-ambiguous cases.

#### 7.4.6. Translator

The translator of Prolog-10 into Toy-Prolog (Appendix A.3, lines 1004–1088) is invoked by the call

```
translate( SourceFileName, OutputFileName ).
```

Commands are translated and also executed (deterministically), so that, for example, a declaration of an infix functor affects subsequent parts of the input program. The translator terminates (and succeeds) after reading in the unary clause

```
end.
```

The program is quite easy to understand. Only the procedure *lookup* may require an explanation. The table pairs variables of the clause with consecutive integers, starting from 0. A variable is a key, so we must use the built-in procedure *eqvar* to locate variables already present in the table. The third parameter of the procedure *lookup* indicates the last number encountered (initially,  $-1$ ), so that only a new variable requires one addition. A more simple-minded solution would be to keep only variables in the table, and count them during lookup. This would require at least  $(n - 1) * n/2$  additions for a clause with  $n$  variables. (In Toy, integers are implemented in a particularly simple way, so this might fill the copy stack with many dead integers). Another possibility is to apply the procedure *numbervars*—inside *put*—to Head and Body jointly.

The translator outputs bare translations. It would be helpful to have source comments transferred to the translation, and to get source variable names paired with numbers (see Section 7.4.1). Try this exercise for yourself.



---

## 8 TWO CASE STUDIES

---

### 8.1. PLANNING

We shall consider planning with respect to a finite, usually small, set of **objects** to which simple **actions** from a finite, and also small, set are applicable. Objects constitute a closed “**world**”. The **state** of the “world” is, by definition, the set of all **relationships** that hold between its objects; we also call these relationships **facts** about objects. As a result of an action, some relationships cease or begin to hold; we say that an action **deletes** or **adds** facts. A fact established by an action is also called a **goal achieved** by this action. Every action transforms one state into another. **Planning** consists in finding a sequence of actions that lead from a given **initial state** to a given **final state**.

As an example, we shall describe one of the so-called cube worlds. There are three cubes, *a*, *b*, *c*, and *floor*. All we can do with them is stack cubes on cubes or on the floor. There are two types of facts concerning a cube *U* and an object *W*: *U* is sitting on *W*, and *U* is clear (this means that nothing is sitting on *U*). The set of possible states is determined by naming all meaningless (i.e. impossible or forbidden) combinations of facts:

- A cube *X* sitting on a *clear* cube *Y*;
- A cube sitting on two different objects;
- Two different cubes sitting on the same cube;
- An object sitting on itself.

There is one kind of action: move a single clear block, either from another block onto the floor, or from an object onto another clear block (the object must differ from both blocks). As a result of moving *X* from *Y* onto *Z*, *X* is sitting on *Z* instead of *Y*, *Y* is clear (unless it is the floor), *Z* is not clear (unless it is the floor).



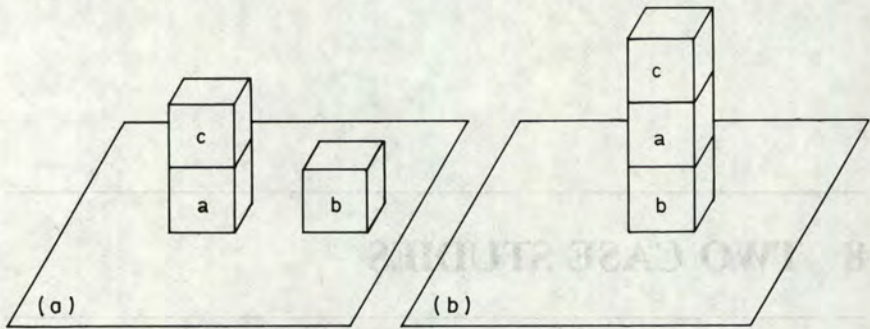


FIG. 8.1 (a) An initial state of the cubes world. (b) A final state of the cubes world.

Even in this microscopic world, planning may require some sophistication. It is reasonable to postulate that a desirable fact, once added, will never be deleted (otherwise we risk an infinite loop). However, let the initial state be that of Fig. 8.1a, described by a conjunction of five facts:

a on floor, b on floor, c on a, clear(b), clear(c).

Let the final state be that of Fig. 8.1b, described by a conjunction of two goals: c on a, a on b. The first goal is trivially achieved. To put a on b, though, we must remove c from a, i.e. destroy an already achieved goal. The simple strategy of achieving goals one by one (and freezing all relevant facts) would not work in this case.

In a more crowded "world", a state might comprise so many facts that its direct representation (as a list, say) would be impractical. Moreover, even a small change might require copying large data structures. Clausal representation is free from this disadvantage but it is unwieldy when a change must be undone, and of course planning is a trial-and-error process. What we need is a method of incrementally describing incremental changes, and making them easily undoable.

A state and an action determine the next state, if we assume that the action does not affect facts not mentioned explicitly in the description of the action's effects as added or deleted. Given an initial state and a plan, i.e. a sequence of actions, we can check whether a fact holds in the resulting final state. To undo an action, we remove it from the plan (in practice, this may be slightly more complicated).

For any particular planning problem, the initial state can be considered fixed. The final state should be given implicitly, as a conjunction of facts to be established by a plan we are going to find. This approach was taken by D. H. D. Warren in his remarkable planning program, WARPLAN.



In WARPLAN, a world description is separated from the planning procedure (see Listing 8.1, pp. 221–223, lines 1–26, for the description of our cube world). Objects are given implicitly, in descriptions of actions and facts. Actions are defined by three procedures. The two-parameter procedure

`can( Action, Precondition )`

serves as a catalogue—one clause per action; Precondition is a conjunction of facts that must hold for Action to be applicable. A conjunction is either a fact, or a pair of conjunctions constructed by the infix functor `&`, e.g. `c on a & a on b`.

Two other procedures,

`add( Fact, Action )`

`del( Fact, Action )`

give facts added and deleted by available actions (and, conversely, actions which can add or delete a fact). Impossible combinations of facts are listed in the procedure

`imposs( Conjunction )`

In these four procedures, we can use variables instead of world objects to express general laws, e.g. “a clear cube U is sitting on a cube V”:

`U on V & notequal( V, floor ) & clear( U )`

For efficiency, facts that hold in the initial state, and are unaffected by any action, are listed in the procedure

`always( Fact )`

Other facts that hold in the initial state are supplied by the procedure

`given( InitialStateName, Fact )`

The initial state is denoted by its name, e.g. *start*. A state derived from it by actions  $A_1, \dots, A_n$  is denoted by the term

`InitialStateName : A1 :  $\dots$  : An,`

e.g.

`start : move( c,a, floor ) : move( a, floor, b ) : move(c, floor, a)`

The planning program (Listing 8.2, pp. 224–226) operates independently of specific world descriptions. It assumes the presence of an appropriate data base whose coherence is the responsibility of the user.



The program begins with a conjunction of facts (i.e. the description of a desired final state) and the empty plan. In each step, the conjunction shrinks and/or the plan grows; successive intermediate states approximate the final state. Roughly speaking, the plan is constructed backwards: we look for preconditions of actions that achieve the final state, then for preconditions of actions that achieve those preconditions, etc. Unless a fact holds in an intermediate state, the program chooses an action that adds this fact, inserts the action into the current partial plan, removes the fact from the current conjunction and adds to it the action's preconditions.

A partial plan usually contains variables. For example, to achieve a on b, we use the action *move(a, V, b)*, whose precondition includes the fact a on V (for an unknown V). Such variables require some care: the fact U on c may, in general, differ from a on V, even though the two terms are unifiable. We can either use the built-in procedure `==` to compare facts, or temporarily instantiate their variables (by the built-in procedure *numbervars*) prior to the comparison.

In addition to the current conjunction and plan, the program maintains a conjunction of desirable facts already planned for. No newly inserted action can destroy any of these *preserved* facts.

The program is amazingly concise. In Warren's original paper it was accompanied by many pages of detailed considerations. Hence, the absence of proper comments in the program text. Below we shall present, in our own words, some indispensable technical explanations.

The main planning routine, *plan*, is called only if the final state description is not inconsistent (lines 10–13), i.e. if it does not imply one of the impossible combinations of facts. *plan* has three input parameters—facts to be achieved, facts already achieved (initially *true*; see line 13) and the current plan—and one output parameter, the final plan. The procedure *solve* is called for each fact of the initial goal list (see lines 30–32). It has five parameters: a fact to be established, preserved facts, the current plan, preserved facts after *solve* has succeeded and the new plan.

Every clause of *solve* accounts for a different status of the fact (lines 35–39). It may be always true; it may be true by virtue of general laws external to "worlds" (e.g. equality or inequality of objects will be checked by this clause); it may hold in the state described by the current plan (to preserve it, we add it to the facts planned for; see lines 83–84); otherwise (the last clause) we choose an action and call *achieve*.

The procedure *achieve* (lines 41–49) tries to apply a given action, i.e. to insert it into the current plan (as the last action, or as the last but one, etc.). The action U is applicable if it deletes none of the preserved facts, and if its precondition is consistent with these facts and if a plan for



achieving this precondition can be constructed. Notice that possible additions to *P* (preserved facts) made by the recursive call on *plan* are invisible to *achieve*: they are only needed “locally” during the construction of the intermediate plan *T1*. The additional call on *preserves* (line 45) is necessary because of variables in the plan. For example, the action *move(b, a, W)* need not delete the fact *clear(c)*, so *preserves* lets it through; however, *plan* may instantiate *W* as *c*, and this ought to cause a failure.

If, for any of these reasons, the action *U* cannot be added at the end of the plan, *achieve* will try to undo the last action *V* and insert *U* earlier into the plan. This is only possible if *V* does not delete the fact to be added by *U*. The procedure *retrace* (lines 65–73) removes from the set of preserved facts all facts that may be established by *V* but are different from *V*’s preconditions. Specifically, it removes the facts added by *V* (lines 68–69) and the facts that constitute the precondition of *V* (lines 70–71)—the latter facts will be re-inserted by *append* (see lines 66, 86–87)<sup>1</sup>.

A few comments on the remaining procedures. A fact holds after executing a given plan (lines 52–55), if it is *given* or added by one of the actions, and preserved by all subsequent actions (if any). Two conjunctions, *C* and *P*, are inconsistent (lines 76–78, 93–97) if *C&P* contains all facts of an impossible combination *S*, except those which—like *not-equal*—are tested “metaphysically” (see line 95). For disjoint *C*, *S* this cannot be the case—hence the call on *intersect* which is relatively cheap. Two object descriptions *X* and *Y*, with variables instantiated by *number-vars* in *mkground* (line 101), may refer to the same object if *X* = *Y* or *X* = ‘*V*’(–) or *Y* = ‘*V*’(–)—see line 99. The procedure *elem* (lines 89–91) extracts single facts from a nested conjunction; it can be used both to test membership, and to generate facts.

Now that you have acquainted yourself with the planning program, try it on a richer world. Here is the world of a robot that walks around several rooms, moves some boxes, etc. (see Listing 8.1, lines 33–101). Figure 8.2 depicts an initial state of this world. There are six points, five rooms, four doors, three boxes, a light switch, and the robot. Nine types of facts are considered: *at*(Object, Point), *on*(Object, Box), *nextto*(Object1, Object2), *pushable*(Object), *inroom*(Object, Room), *locinroom*(Point, Room), *connects*(Door, Room1, Room2), *status*(Lightswitch, OnOff), *onfloor*—the latter characterizes the robot. Only the robot performs actions—there are seven of them (see lines 64–77).

<sup>1</sup> The special treatment of *V*’s preconditions is necessary for actions which add facts listed among their own preconditions. If *retrace* simply deleted *V*’s effects, such preconditions could be lost from the list of facts which must be preserved by *U*, and those parts of the plan which achieve “locally desirable” goals could inadvertently be destroyed in the insertion process.



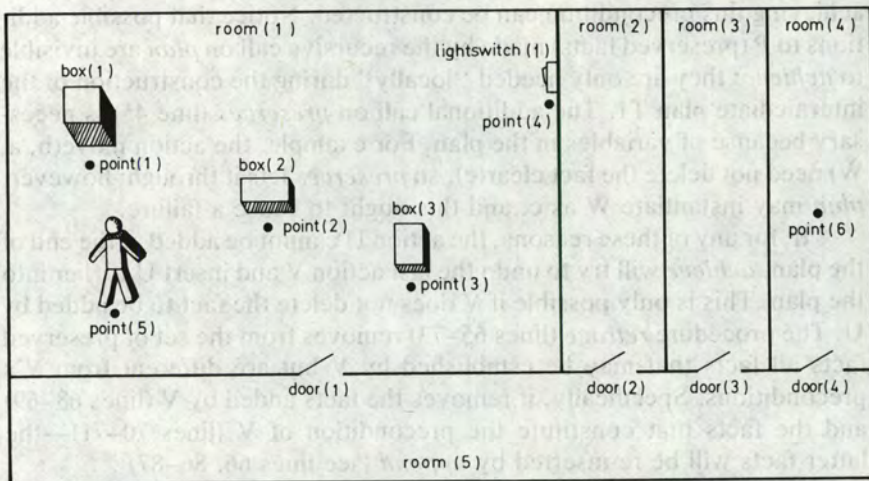


FIG. 8.2 "STRIPS" world.

The procedure *del* merits a comment. It is supposed to delete more than it should—we count on *add* to straighten the situation out. For example, the action *turnon(S)* removes whatever status of *S* may be recorded (line 54); “a moment later” it adds the appropriate fact (line 39). The clauses in lines 49–50 say that a moved object *X* is no longer “next to” anything. However, this does not apply to the robot manipulating a box (lines 46–48)—*del* fails, i.e. the fact is *not* deleted.

For sample results, see Listing 8.3, p. 227.

Although WARPLAN is a feat of ingenuity, there is much more to planning than it does account for. For one thing, the plans it generates need not be optimal, i.e. contain the least possible number of actions. For example, action *U* in *achieve* (lines 47–49) is executed when it preserves *V*'s precondition *P*; if we checked that *U* establishes *P*, we might delete actions which had been planned to establish it. A much more profound problem: in general, it is likely that conditional or iterative plans will be required, rather than sequential (the robot explores the world).

Even with these (and other) limitations, and despite exponential time complexity, WARPLAN is an excellent tool for experiments with rigorous world descriptions. One example is the world of a robot that assembles cars. Warren has also demonstrated how his program can be used to compile arithmetic expressions into machine code (the code is treated as a plan for placing some values in some registers).



# LISTING 8.1 WARPLAN—Examples of worlds.

```

1  % % % % % WARPLAN - cube worlds
2
3  :- op( 50, xfx, on ).
4
5  add( U on W, move( U, V, W ) ).
6  add( clear( V ), move( U, V, W ) ).
7
8  del( U on Z, move( U, V, W ) ).
9  del( clear( W ), move( U, V, W ) ).
10
11  can( move( U, V, floor ),
12      U on V & notequal( V, floor ) & clear( U ) ).
13  can( move( U, V, W ),
14      clear( W ) & U on V & notequal( U, W ) & clear( U ) ).
15
16  imposs( X on Y & clear( Y ) ).
17  imposs( X on Y & X on Z & notequal( Y, Z ) ).
18  imposs( X on Z & Y on Z & notequal( Z, floor ) & notequal( X, Y ) ).
19  imposs( X on X ).
20
21  % The three blocks problem.
22  given( start, a on floor ).
23  given( start, b on floor ).
24  given( start, c on a ).
25  given( start, clear( b ) ).
26  given( start, clear( c ) ).
27
28  :- plans( c on a & a on b, start ).
29  :- plans( a on b & b on c, start ).
30  :- delop( 'on' ), redefine.
31  % -----
32
33  % % % % % WARPLAN - the STRIPS problem
34
35  add( at( robot, P ), goto1( P, R ) ).
36  add( nextto( robot, X ), goto2( X, R ) ).
37  add( nextto( X, Y ), pushto( X, Y, R ) ).
38  add( nextto( Y, X ), pushto( X, Y, R ) ).
39  add( status( S, on ), turnon( S ) ).
40  add( on( robot, B ), climbon( B ) ).
41  add( onfloor, climboff( B ) ).
42  add( inroom( robot, R2 ), gothrough( D, R1, R2 ) ).
43
44  del( at( X, Z ), U ) :- moved( X, U ).
45  del( nextto( Z, robot ), U ) :- !, del( nextto( robot, Z ), U ).
46  del( nextto( robot, X ), pushto( X, Y, R ) ) :- !, fail.
47  del( nextto( robot, B ), climbon( B ) ) :- !, fail.
48  del( nextto( robot, B ), climboff( B ) ) :- !, fail.
49  del( nextto( X, Z ), U ) :- moved( X, U ).
50  del( nextto( Z, X ), U ) :- moved( X, U ).
51  del( on( X, Z ), U ) :- moved( X, U ).
52  del( onfloor, climbon( B ) ).
53  del( inroom( robot, Z ), gothrough( D, R1, R2 ) ).
54  del( status( S, Z ), turnon( S ) ).
55
56  moved( robot, goto1( P, R ) ).
57  moved( robot, goto2( X, R ) ).
58  moved( robot, pushto( X, Y, R ) ).
59  moved( X, pushto( X, Y, R ) ).
60  moved( robot, climbon( B ) ).

```



# LISTING 8.1 (Continued)

```

61 moved( robot, climboff( B ) ).
62 moved( robot, gothrough( D, R1, R2 ) ).
63
64 can( goto1( P, R ),
65 locinroom( P, R ) & inroom( robot, R ) & onfloor ).
66 can( goto2( X, R ),
67 inroom( X, R ) & inroom( robot, R ) & onfloor ).
68 can( turnon( lightswitch(S) ),
69 on( robot, box(1) ) & nextto( box(1), lightswitch(S) ) ).
70 can( pushto( X, Y, R ),
71 pushable( X ) & inroom( Y, R ) & inroom( X, R ) &
72 nextto( robot, X ) & onfloor ).
73 can( gothrough( D, R1, R2 ),
74 connects( D, R1, R2 ) & inroom( robot, R1 ) &
75 nextto( robot, D ) & onfloor ).
76 can( climboff( box(B) ), on( robot, box(B) ) ).
77 can( climbon( box(B) ), nextto( robot, box(B) ) & onfloor ).
78
79 always( inroom( D, R1 ) ) :- always( connects( D, R1, R2 ) ).
80 always( connects( D, R2, R1 ) ) :- connects1( D, R1, R2 ).
81 always( connects( D, R1, R2 ) ) :- connects1( D, R1, R2 ).
82 always( pushable( box(N) ) ).
83 always( locinroom( point(N), room(1) ) ) :- range( N, 1, 5 ).
84 always( locinroom( point(6), room(4) ) ).
85 always( inroom( lightswitch(1), room(1) ) ).
86 always( at( lightswitch(1), point(4) ) ).
87
88 connects1( door(N), room(N), room(5) ) :- range( N, 1, 4 ).
89
90 range( M, M, _ ).
91 range( M, L, N ) :-
92 L < N, L1 is L + 1, range( M, L1, N ).
93
94 imposs( at( X, Y ) & at( X, Z ) & notequal( Y, Z ) ).
95
96 given( strips1, at( box(N), point(N) ) ) :- range( N, 1, 3 ).
97 given( strips1, at( robot, point(5) ) ).
98 given( strips1, inroom( box(N), room(1) ) ) :- range( N, 1, 3 ).
99 given( strips1, onfloor ).
100 given( strips1, status( lightswitch(1), off ) ).
101 given( strips1, inroom( robot, room(1) ) ).
102
103 % A few tests.
104 :- plans( at( robot, point(5) ), strips1 ).
105 :- plans( at( robot, point(1) ) & at( robot, point(2) ), strips1 ).
106 :- plans( at( robot, point(4) ), strips1 ).
107 :- plans( status( lightswitch(1), on ), strips1 ).
108 :- plans( at( robot, point(6) ), strips1 ).
109 :- plans( nextto( box(1), box(2) ) &
110         nextto( box(3), box(2) ), strips1 ).

```



## LISTING 8.2 WARPLAN—The general planner

```

1  % % % % %      WARPLAN - A System for Generating Plans
2  % % % % %      ( published with the kind permission of the Author,
3  % % % % %      David H.D.Warren )
4
5  % % % The general planner.
6  % -----
7  :- op( 200, xfy, & ), op( 100, yfx, : ).
8
9  % Generate and output a plan.
10 plans( C, _ ) :-
11 inconsistent( C, true ), !, write( 'Impossible.' ), nl.
12 plans( C, T ) :-
13 plan( C, true, T, T1 ), output( T1 ), !.
14 plans( _, _ ) :-
15 write( 'Cannot do this.' ), nl.
16
17 output( Xs:X ) :-
18 numbervars( Xs:X, 1, _ ), output1( Xs ), output2( X, ':' ).
19 output( _ ) :- write( 'Nothing need be done.' ), nl.
20
21 output1( Xs:X ) :- !, output1( Xs ), output2( X, ':' ).
22 output1( X ) :- output2( X, ':' ).
23
24 output2( Item, Punct ) :- write( Item ), write( Punct ), nl.
25
26 % Main planning routine.
27 % Definitions of 'always', 'imposs', 'given', 'can', 'add', 'del' -
28 % see specific world descriptions.
29
30 plan( X&C, P, T, T2 ) :-
31 !, solve( X, P, T, P1, T1 ), plan( C, P1, T1, T2 ).
32 plan( X, P, T, T1 ) :- solve( X, P, T, _ , T1 ).
33
34 % Ways of solving a goal.
35 solve( X, P, T, P, T ) :- always( X ).
36 solve( X, P, T, P, T ) :- X.
37 solve( X, P, T, P1, T ) :- holds( X, T ), and( X, P, P1 ).
38 solve( X, P, T, X&P, T1 ) :-
39 add( X, U ), achieve( X, U, P, T, T1 ).
40
41 % Methods of achieving a goal -
42 % by extension:
43 achieve( _, U, P, T, T1:U ) :-
44 preserves( U, P ), can( U, C ), not inconsistent( C, P ),
45 plan( C, P, T, T1 ), preserves( U, P ).
46 % by insertion:
47 achieve( X, U, P, T:V, T1:V ) :-
48 preserved( X, V ), retrace( P, V, P1 ),
49 achieve( X, U, P1, T, T1 ), preserved( X, V ).
50
51 % Check if a fact holds in a given state.
52 holds( X, :V ) :- add( X, V ).
53 holds( X, T:V ) :-
54 !, preserved( X, V ), holds( X, T ), preserved( X, V ).
55 holds( X, T ) :- given( T, X ).
56
57 % Prove that an action preserves a fact.
58 preserves( U, X&C ) :- preserved( X, U ), preserves( U, C ).
59 preserves( _, true ).

```



## LISTING 8.2 (Continued)

```

60
61 preserved( X, V ) :- check( pres( X, V ) ).
62 pres( X, V ) :- mkground( X&V ), not del( X, V ).
63
64 % Retracing a goal already achieved.
65 retrace( P, V, P2 ) :-
66     can( V, C ), retrace( P, V, C, P1 ), append( C, P1, P2 ).
67
68 retrace( X&P, V, C, P1 ) :-
69     add( Y, V ), X == Y, !, retrace( P, V, C, P1 ).
70 retrace( X&P, V, C, P1 ) :-
71     elem( Y, C ), X == Y, !, retrace( P, V, C, P1 ).
72 retrace( X&P, V, C, X&P1 ) :- retrace( P, V, C, P1 ).
73 retrace( true, _, _ , true ).
74
75 % Inconsistency with a goal already achieved.
76 inconsistent( C, P ) :-
77     mkground( C&P ), imposs( S ),
78     check( intersect( C, S ) ), implied( S, C&P ), !.
79
80 % % % Utilities.
81 % -----
82
83 and( X, P, P ) :- elem( Y, P ), X == Y, !.
84 and( X, P, X&P ).
85
86 append( X&C, P, X&P1 ) :- !, append( C, P, P1 ).
87 append( X, P, X&P ).
88
89 elem( X, Y&_ ) :- elem( X, Y ).
90 elem( X, _&C ) :- !, elem( X, C ).
91 elem( X, X ).
92
93 implied( S1&S2, C ) :- !, implied( S1, C ), implied( S2, C ).
94 implied( X, C ) :- elem( X, C ).
95 implied( X, _ ) :- X.
96
97 intersect( S1, S2 ) :- elem( X, S1 ), elem( X, S2 ).
98
99 notequal( X, Y ) :- not X=Y, not X='V'(_), not Y='V'(_).
100
101 mkground( X ) :- numbervars( X, 0, _ ).

```

Toy-Prolog listening:

```
?- % files without test cases but with end. at the end
:- consult( planner ), consult( cubes ).
?-
:- plans( c on a & a on b, start ).
start :
move( c, a, floor ) :
move( a, floor, b ) :
move( c, floor, a ).
?-
:- plans( a on b & b on c, start ).
start :
move( c, a, floor ) :
move( b, floor, c ) :
move( a, floor, b ).
?-
:- delop( 'on' ), reconsult( strips ).
?-
:- plans( at( robot, point(5) ), strips1 ).
Nothing need be done.
?-
:- plans( at( robot, point(1) ) & at( robot, point(2) ), strips1 ).
Impossible.
?-
:- plans( at( robot, point(4) ), strips1 ).
strips1 :
goto1( point( 4 ), room( 1 ) ).
?-
:- plans( status( lightswitch(1), on ), strips1 ).
strips1 :
goto2( box( 1 ), room( 1 ) ) :
pushto( box( 1 ), lightswitch( 1 ), room( 1 ) ) :
climbon( box( 1 ) ) :
turnon( lightswitch( 1 ) ).
?-
:- plans( at( robot, point(6) ), strips1 ).
strips1 :
goto2( door( 1 ), room( 1 ) ) :
gothrough( door( 1 ), room( 1 ), room( 5 ) ) :
goto2( door( 4 ), room( 5 ) ) :
gothrough( door( 4 ), room( 5 ), room( 4 ) ) :
goto1( point( 6 ), room( 4 ) ).
?-
:- plans( nextto( box(1), box(2) ) & nextto( box(3), box(2) ), strips1 ).
strips1 :
goto2( box( 1 ), room( 1 ) ) :
pushto( box( 1 ), box( 2 ), room( 1 ) ) :
goto2( box( 3 ), room( 1 ) ) :
pushto( box( 3 ), box( 2 ), room( 1 ) ).
?- stop.
```

Toy-Prolog, end of session.



## BIBLIOGRAPHIC NOTES

WARPLAN is described in Warren (1974). Our presentation has been greatly influenced by this excellent paper. The program we publish here is a slightly cleaned-up version of the text given in Coelho *et al.* (1980), where all the mentioned examples of worlds can also be found. The robot's world was introduced by Fikes and Nilsson (1971) as a test case for their system STRIPS; Warren (1974) used it to compare the performance of the two systems. An extension of WARPLAN, intended for generating conditional plans, was described in Warren (1976).

## 8.2. PROLOG AND RELATIONAL DATA BASES

In this section, we shall be primarily concerned with data bases in the limited sense: a **data base** is a purposefully structured collection of stored data, often pertaining to an organisation (e.g. a bank, factory, university, warehouse). In a **relational** data base all data are conceptually grouped into relations, which are usually depicted as rectangular tables as in Fig. 8.3. A column in the table is called an **attribute** and referred to by a name, e.g. *dno*. All values of an attribute belong to a common **domain**, e.g. each salary belongs to integers. A relation is a set of **tuples** (table rows) which

	empno	name	dno	salary	mgrno
EMP	13	Miller	0	1500	19
	21	Jones	1	1000	13
	35	Brown	1	1000	21
	38	White	1	800	35
	43	Smith	2	1200	13
	61	Thomas	1	850	21
	89	Morgan	1	1050	35
	42	Miller	1	850	35

	dno	name	mgrno
DEPT	1	PublicRelations	21
	2	Security	43

FIG. 8.3 Contents of a relational data base.



consist of attribute values, e.g.

< 38, White, 1, 800, 35 >.

Tuples belong to the set described by a **relation schema** which specifies names, domains and order of attributes, e.g.

EMP < integer empno, string name, integer dno,  
integer salary, integer mgrno >

DEPT < integer dno, string name, integer mgrno >

Two tuples may share the value of an attribute, and thus implicitly fall into one group; for example, Brown and Thomas are both subordinates of a manager whose number is 21.

A relation can be changed by inserting, deleting or updating some of its tuples. These operations are referred to as **data manipulation**.

A query to the data base is answered by enumerating tuples of the resulting relation (or by computing an **aggregate function**, such as "average" or "total", over these tuples). Most queries are expressible in terms of the following primitive operations on relations. 2

- Selection** chooses tuples for which a given condition holds; for example, we can select from EMP those employees of department 1 who earn over 900 (there are three such tuples).
- Projection** neglects some attributes and (possibly) reorders the remaining ones; for example, we can project EMP over *name*, *empno*, and *salary*, to get

< Miller, 13, 1500 >

and seven other triples.

- Join** of two relations A, B forms a new relation. It consists of those concatenations of tuples from A with tuples from B, for which a given condition holds. For example, the join of EMP and DEPT, such that department numbers coincide, consists of the tuple

< 21, Jones, 1, 1000, 13, 1, PublicRelations, 21 >

and six other 8-tuples.

- Unconditional join is a **product**; for EMP and DEPT the product consists of sixteen 8-tuples.
- Finally, set operations, namely **union**, **intersection** and **difference**, can be applied to two relations whose corresponding attributes belong to the same domain, i.e. whose schemata differ only in names.

Much of this conceptual framework is naturally translated into Prolog. A relation is modelled as a procedure made of unit clauses which



correspond to tuples, for example:

'EMP'( 13, 'Miller', 0, 1500, 19 ).

'EMP'( 21, 'Jones', 1, 1000, 13 ).

etc.

(we use quotes to prevent capitalized names from being treated as variables). To change a relation, we use the built-in procedures *assert* and *retract*.

Primitive operations on relations are expressed in terms of procedure calls. For example, the procedure

s( Empno, Name, Dno, Salary, Mgrno ) :-

'EMP'( Empno, Name, Dno, Salary, Mgrno ),

Dno = 1, Salary > 900.

can be used to generate all tuples for employees of department 1 who earn over 900 (i.e. to implement selection):

:- s( E, N, D, S, M ), write( ( E, N, D, S, M ) ), nl, fail.

Better still, we can substitute 1 for Dno and remove the test:

s( E, N, 1, S, M ) :- 'EMP'( E, N, 1, S, M ), S > 900.

The procedure *p* can be used to implement projection:

p( Name, Empno, Salary ) :-

'EMP'( Empno, Name, \_, Salary, \_ ).

The composition of these two operations can be expressed in Prolog quite succinctly:

s\_then\_p( Name, Empno, Salary ) :-

'EMP'( Empno, Name, 1, Salary, \_ ), Salary > 900.

Or we can put this directly into a query:

:- 'EMP'( E, N, 1, S, \_ ), S > 900,

write( ( N, E, S ) ), nl, fail.

Finally, here is the join of EMP and DEPT over coinciding department numbers:

j( Empno, NameE, DnoE, Salary, MgrnoE, DnoD, NameD, MgrnoD ) :-

'EMP'( Empno, NameE, DnoE, Salary, MgrnoE ),

'DEPT'( DnoD, NameD, MgrnoD ).



All these operations are neatly explained in terms of static interpretation of procedures (try for yourself!). Set operations are even more straightforward. Let  $a(X_1, \dots, X_n)$  and  $b(X_1, \dots, X_n)$  denote generators of tuples, such as 'DEPT'( D, N, M ) or  $p(N, E, S)$ . We have

```
aUNIONb(  $X_1, \dots, X_n$  ) :-
    a(  $X_1, \dots, X_n$  ) ; b(  $X_1, \dots, X_n$  ).
aINTERSECTIONb(  $X_1, \dots, X_n$  ) :-
    a(  $X_1, \dots, X_n$  ), b(  $X_1, \dots, X_n$  ).
aDIFFERENCEb(  $X_1, \dots, X_n$  ) :-
    a(  $X_1, \dots, X_n$  ), not b(  $X_1, \dots, X_n$  ).
```

Queries which involve only primitive operations can be answered without actually creating the resulting relation. Its tuples can be generated by a failure-driven loop and displayed immediately. To compute an aggregate function, however, we need the whole attribute (column) at once. We can construct it by means of the procedure *bagof* (see Section 4.2.4); for example:

```
:- bagof( Salary, 'EMP'( _, _, _, Salary, _ ), Salaries ),
   max_of( Salaries, MaxSal ), write( MaxSal ), nl.
```

Sometimes we can also use *bagof* for efficiency. For example, we look for employees of department 1 who earn no more than a 1000 and who are FTU members since at least 1980:

```
:- 'EMP'( E, N, 1, S, _ ), S =< 1000,
   'FTU'( E, _, _, DateJoined ), DateJoined =< 1980,
   write( ( E, N ) ), nl, fail.
```

With those implementations of Prolog which do not support clause indexing, the entire relation FTU would be scanned many times. Instead, we can precompute the necessary set:

```
:- bagof( Empno, ( 'FTU'( Empno, _, _, DateJoined ),
                  DateJoined =< 1980 ), FTUMembers ),
   'EMP'( E, N, 1, S, _ ), S =< 1000,
   member( E, FTUMembers ), write( ( E, N ) ), nl, fail.
```

There are queries which cannot be expressed as a composition of selections, projections, joins and aggregate functions. A classical example: find every employee who earns more than at least one of her/his superiors. The relation "is a superior of" is inherently transitive, but we can only express the relations "is an immediate manager of", "is an immediate manager of an immediate manager of" etc. In Prolog, how-



ever, the problem is easily solved. For example, we can define a procedure to generate managers' salaries:

```
mgr_sal( Mgrno, Salary ) :- 'EMP'( Mgrno, _, _, Salary, _ ).
mgr_sal( Mgrno, Salary ) :-
    'EMP'( Mgrno, _, _, _, MgrMgrno ),
    mgr_sal( MgrMgrno, Salary ).
```

(try to rewrite it so as to avoid the repeated pass through EMP). From the standpoint of the caller, this generator is indistinguishable from those made of unit clauses. The query can be written as follows:

```
:- 'EMP'( Empno, _, _, Salary, Mgrno ),
   once( ( mgr_sal( Mgrno, MgrSal ), MgrSal < Salary ) ),
   write( Empno ), nl, fail.
```

A relation which is computed rather than stored (e.g. *s*, *p*, *s\_then\_p*, *j* above) is called a **view** in the relational data base terminology. A view results from primitive operations on stored relations—also indirectly, via other views—and it changes as those relations change (and conversely, a change in a view might influence those relations—but this poses quite nontrivial problems). The relation *mgr\_sal*, however, can only be obtained by embedding primitive operations in a host programming language, furnished with recursion or iteration. An important advantage of Prolog is its ability to express tuples, views, and special programs in the same language. In particular, it offers a possibility of enforcing **integrity constraints**—application-specific conditions of the coherence of data. Constraints should be tested prior to any change to a relation. For example, we can use this procedure to insert only correct tuples:

```
insert( Tuple ) :-
    correct_insert( Tuple ), !, assertz( Tuple ).
insert( Tuple ) :- signal_violation( Tuple ).
correct_insert( 'EMP'( E, _, D, _, M ) ) :-
    !, E \= M, 'DEPT'( D, _, _ ). % there is such a dept
correct_insert( _ ). % others are OK
```

On the whole, Prolog is a powerful tool for data base applications. Admittedly, there is more to data base systems than our presentation suggests. For one thing, the size of a real data base may far exceed the capacity of any existing Prolog implementation. The model described in Chapter 6 ought to be augmented: clauses would be stored on disk and handled by standard or specialized access methods. Second, every practical data base implementation should address problems such as concurrent



execution of users' commands, recovery after hardware failures, etc., etc. There are no ready solutions in Prolog but presumably they can be programmed into it.

Surprisingly, Prolog is, in a sense, too strong, too unrestricted. For example, to ensure the conformance of a tuple with the relation schema, some form of type checking is required, presumably as explicit tests. Unrestrained use of assert/retract may also ruin the integrity of a data base in other ways. Consequently, Prolog should rather be considered a tool for implementing more restricted user interfaces: queries and commands in a user language are analysed (types checked, integrity ensured, etc.), and only then translated into Prolog.

A particularly attractive option would be to query the data base in a natural language. Several encouraging small experiments have been carried out. At the moment, though, this is much more a research problem in its own right than a generally available programming technique.

Relational data languages, notably Sequel and Query-by-Example, provide syntactic sugar for relation schema definitions, data manipulation and queries (involving Boolean expressions and compositions of primitive relational operations). In contrast with natural language interfaces, a Prolog implementation of a relational data language is a programming task of moderate complexity.

We shall now present Toy-Sequel, a relational data language patterned after Sequel and implemented in Prolog (see Listing 8.4). With the exception of aggregate functions, expressions in tuple specifications and some exotic features, it supports all that is essential in Sequel. Extensions are relatively easy to introduce (we left them out to make the program shorter). To give the flavour of the language, here is an annotated conversation with our program, initiated by the call

```
:- toysequel.
```

To begin with, we specify a few relation schemas:

```
create EMP < string name, integer salary, integer dno >.
create DEPT < integer dno, string manager >.
create BoardMembers < string name, string position,
                      integer seniority >.
```

Now we insert some tuples:

```
into EMP insert < "Brown", 1000, 1 >, < "White", 800, 1 >,
               < "Miller", 850, 1 >, < "Barry", 900, 2 >,
               < "Thomas", 850, 1 >, < "Morgan", 1050, 1 >.
into DEPT insert < 1, "Jones" >, < 2, "Smith" >.
```



We can ask what relations the data base contains; Toy-Sequel displays their names (its responses are italicized):

relations.

*BoardMembers*

*DEPT*

*EMP*

What is the schema of EMP?

relation EMP.

*string name*

*integer salary*

*integer dno*

A select expression determines a set of tuples. They may be displayed. For example, who in departments other than 2 earns at least 1000?

select from EMP tuples < name, salary >  
where dno <> 2 and salary >= 1000.

*Brown 1000*

*Morgan 1050*

Or they may be inserted elsewhere:

into EMP insert

select from DEPT tuples < manager, 1000, dno >.

In the absence of "where ...", the condition is taken as true.

Both managers have the salary 1000. We can give Smith a raise:

update EMP so that salary = 1200 where name = "Smith".

Fire Barry:

from EMP delete tuples where name = "Barry".

If several relations are involved, e.g. in a join, attribute names may be ambiguous. To disambiguate, qualify them with relation names. For example:

select from EMP, DEPT tuples < name, EMP\_dno, manager >  
where EMP\_dno = DEPT\_dno.

(Actually, EMP\_dno may be replaced with dno: an unqualified attribute name is qualified with the leftmost appropriate relation name.)

A relation may be accessed in several places at once. For example, to compare salaries of different employees we need the product of EMP by EMP. We must give one of the occurrences an alias name and so allow

unambiguous references to attribute values. The following query joins the relations EMP, DEPT and EMP alias Mgr, to find employees who earn more than their (immediate) manager:

```
select from EMP, DEPT, Mgr = EMP tuples < EMP_name >
  where EMP_dno = DEPT_dno
    and DEPT_manager = Mgr_name
    and Mgr_salary < EMP_salary.
```

*Morgan*

Again, the qualification with EMP is superfluous, as well as the qualification of *manager*.

A similar condition can be used to give a raise of half the difference in salaries to those who earn over 100 less than their manager:

```
update EMP using DEPT, Mgr = EMP
  so that salary = EMP_salary + ( Mgr_salary - EMP_salary )/2

  where EMP_dno = DEPT_dno
    and manager = Mgr_name
    and Mgr_salary > EMP_salary + 100.
```

Two miscellaneous queries. Find employees whose names do not begin with M:

```
select from EMP tuples < name >
  where name < "M" or name >= "N".
```

*pos. lehrsyhage.*

(five of them). And find EMP tuples with a nonexistent department number—this is a kind of (manual...) integrity checking:

```
select from EMP tuples < name, salary, dno > where
  not < dno > in select from DEPT tuples < dno >.
```

*in* denotes set membership. The name *dno* in the nested select expression pertains to DEPT.

Time to finish. The relation BoardMembers will not be necessary, after all:

cancel BoardMembers.

Store the data base in a file:

dump to AAA.

(next time we shall begin with

load from AAA.



and resume at this point). Finally, return control to Prolog:  
 stop.

We shall not go into details of the Toy-Sequel interpreter. The rationale for its design was given above; the program is (almost) self-documenting. The following remarks account for a few central technical decisions.

The main procedure, *toysequel* (lines 4–7 in the listing), repeatedly reads and executes commands. The procedure *getcommand* (lines 9–11) returns a Prolog goal, which is the translation of a command, and a flag. The flag remains uninstantiated if the command is correct, otherwise it is instantiated as *error*. The procedure *docommand* (lines 13–14) executes a correct command's translation, and does nothing in the case of errors.

A command is processed in three phases. The text, terminated with a dot, is read in (lines 32–43) and then passed through a scanner, implemented as a metamorphosis grammar (lines 45–84). It classifies tokens as names, strings, integers and single non-alphanumeric characters. A list of tokens goes to the command compiler—a metamorphosis grammar which is the core of the interpreter. The grammar consists of 11 parts, one for each Toy-Sequel command (see lines 113–123).

All commands, except *load* and *stop*, manipulate the relation catalogue. The catalogue is implemented as a three-parameter procedure 'r e l', with a unit clause for each relation schema. A schema stores the name of a relation, a generator of this relation's tuples and a "frame" of symbol table entries linking attribute names and types to variables in the generator (see lines 131–141). For example, the command

```
create EMP< string name, integer salary, integer dno >.
```

adds the clause

```
'r e l'( 'EMP', ' EMP'( Name, Salary, Dno ),  
  [ attr( name, string, Name ), attr( salary, integer, Salary ),  
    attr( dno, integer, Dno ) ] ).
```

Blanks are added to relation names in generators to make conflicts with other procedures less plausible.

The command processors for *select*, *insert*, *delete* and *update* maintain a symbol table—a stack of frames taken from the catalogue. For example, attribute names in the command

```
select from EMP, Mgr = EMP, DEPT  
  tuples < name, Mgr_dno, manager >  
  where dno = DEPT_dno and manager = Mgr_name  
         and salary > Mgr_salary.
```



will be looked for in the following symbol table (see lines 208–223, and 96–102):

```
[ 'EMP' : [ attr( name, string, NameEMP ),
             attr( salary, integer, SalaryEMP ),
             attr( dno, integer, DnoEMP ) ],
  'Mgr' : [ attr( name, string, NameMgr ),
             attr( salary, integer, SalaryMgr ),
             attr( dno, integer, DnoMgr ) ],
  'DEPT' : [ attr( dno, integer, DnoDEPT ),
             attr( manager, string, ManagerDEPT ) ] ]
```

(Nested select expressions would push their own frames onto this stack—see line 277.)

The product of these three relations will be generated by the following calls retrieved from the catalogue:

```
'EMP'( NameEMP, SalaryEMP, DnoEMP ),
'EMP'( NameMgr, SalaryMgr, DnoMgr ),
'DEPT'( DnoDEPT, ManagerDEPT )
```

The condition will be translated into a Prolog goal (see lines 236–237, 239–362). The goal will be executed immediately after the generators (lines 166–168). Attribute names in the condition will be translated into variables from the symbol table. Thus,

```
salary > Mgr_salary
```

will become

```
SalaryEMP > SalaryMgr
```

The “equalities”

```
DnoEMP = DnoDEPT, ManagerDEPT = NameMgr
```

will be processed at compile time, by binding variables together (line 318), so that actually only six different variables will occur in the generators.

The tuple pattern (lines 225–234) will also contain variables from the symbol table:

```
[ NameEMP, DnoMgr, ManagerDEPT ]
```

One such tuple will be displayed in every step of the failure-driven loop (lines 166–168).

A construction that would certainly benefit from a more detailed explanation is *update*. We shall comment on the example shown in the



listing (lines 396–411):

```
update EMP using DEPT, Mgr = EMP
  so that salary = salary + ( Mgr_salary - salary ) / 5
  where salary < Mgr_salary - 1000 and Mgr_name = manager
  and DEPT_dno = dno and not< Mgr_name > in
    select from BoardMembers tuples < name >.
```

First, two copies of the stack frame are created, and two call patterns (OldTup and NewTup):

```
' EMP'( Name, Salary, Dno )
' EMP'( NewName, NewSalary, NewDno )
```

Now, *makemodlist* creates a raw modification list:

```
[ modif( attr( name, string, Name ), NewName, ModName ),
  modif( attr( salary, integer, Salary ), NewSalary, ModSalary ),
  modif( attr( dno, integer, Dno ), NewDno, ModDno ) ]
```

A symbol table is constructed, first a frame for EMP (note that *old* attribute values will be retrieved and used), next for DEPT and EMP (with the alias name Mgr). During the construction of Modifications (lines 420–421, 410), the raw list is changed by *findmname* (lines 454–465, 434): ModSalary is instantiated as true (line 462) to note that the salary will be modified. Finally, *closemodlist* (lines 447–452, 422) binds together variables that stand for unmodified attributes, i.e. Name with NewName and Dno with NewDno. Also, equalities in lines 398–399 cause two other pairs of variables to be bound together (see line 318).

A comment on error treatment. Incorrect data do not terminate processing. Instead, the procedure *ancestor* instantiates the variable Errflag (lines 5, 484, 488)—this prevents the command from being executed (lines 13–14), but the analysis continues. The grammar rules *synerrc* (lines 486–498) display the troublesome token and the others to its right, and then succeed leaving the token list intact.

In actual use, Toy-Sequel would probably be found too simple. However, many extensions are quite straightforward. As an exercise, try to augment Toy-Sequel with defined views, e.g.

```
view EMP1 < name, salary > as
  select from EMP tuples < name, salary > where dno = 1.
```

Another extension: “wild card” tuple specifications, e.g.

```
select from EMP tuples *.
  ( i.e. tuples < name, salary, dno > )
```



```
select from EMP, DEPT tuples < EMP_*, manager >
  where EMP_dno = DEPT_dno.
(i.e. tuples < name, salary, EMP_dno, manager > ).
```

And aggregate functions, e.g.

```
select from EMP average of < salary >.
```

An example of less straightforward modifications is query optimisation. Consider the command:

```
select from EMP, DEPT tuples < name, salary >
  where dno = DEPT_dno and manager = "Jones".
```

The answer will be generated by the calls

```
' EMP'( Name, Salary, Dno ), ' DEPT'( Dno, "Jones" )
```

which access every EMP tuple, even though only one department is involved. The same set of tuples would be generated by the calls

```
' DEPT'( Dno, "Jones" ), ' EMP'( Name, Salary, Dno )
```

but now other departments' tuples would never be retrieved. This optimisation could speed things up considerably for Prolog implementations with clause indexing.

#### BIBLIOGRAPHIC NOTES

The bibliography on data bases is enormous. We shall only name a few positions relevant to our presentation which (of necessity) has only touched on basic facts. Two widely accepted introductory textbooks on data bases in general are Ullman (1982) and Date (1982). The relational model of data was introduced by Codd (1970) and further elaborated by many, including Codd himself (1979). The most popular relational data languages are probably Quel, used in the data base system INGRES (Stonebraker *et al.* 1976), Sequel, created for the system R (Astrahan 1976; Chamberlin *et al.* 1976) and Query-by-Example (Zloof 1977).

In the proceedings of conferences on logic in data bases (Gallaire and Minker 1978, Gallaire *et al.* 1981) there are, in particular, papers on the role of logic programming in data base theory and applications. The advantages of Prolog (and logic programming at large) for data bases have been advocated by quite a few authors, e.g. Kowalski (1978), Gallaire (1983) and Lloyd (1982). A practical demonstration of Prolog's power is Chat80 (Warren and Pereira 1982; Warren 1981), a system with a natural language interface. Queries in English are translated into Prolog calls; they are similar to those produced by Toy-Sequel, but Chat80 performs



some query optimisation. Several other data base applications with natural language interface are described in Dahl (1977), Coelho (1982), and Filgueiras and Pereira (1983).

Another example of data base application of Prolog is an implementation of Query-by-Example (Neves and Williams 1983; Neves *et al.* 1983). Chomicki and Grudziński (1983) describe a system, based on extendible hashing, that manipulates tuples stored on disk. The system has been designed to support data-base-oriented implementations of Prolog.

The Toy-Sequel interpreter was rewritten as a sized-down version of SPOQUEL, a program which we had written with Włodek Grudziński in early 1982. It helped us through a difficult winter.

## LISTING 8.4 Toy—Sequel interpreter

```

1  % ----- Toy-Sequel interpreter -----
2  % (c) COPYRIGHT 1983 - Feliks Kluzniak, Stanislaw Szpakowicz
3  % Institute of Informatics, Warsaw University
4  toysequel :- write( '-- Toy-Sequel, IIUW Warszawa 1983 --' ), nl,
5             repeat, write( ' ' ), tag( getcommand( Cmd, Errflag ) ),
6             tag( docommand( Cmd, Errflag ) ),
7             Cmd = sequelstop, !.
8
9  getcommand( Cmd, Errflag ) :-
10 readcmd( CmdString ),
11 scan( CmdString, TList ), compile( TList, Cmd ).
12
13 docommand( Cmd, Errflag ) :- var( Errflag ), !, Cmd.
14 docommand( _, _ ).
15
16 scan( CmdString, TList ) :-
17 phrase( tokens( TList ), CmdString ), tracescan( TList ).
18
19 compile( TList, Cmd ) :-
20 phrase( command( Cmd ), TList ), !, tracecompile( Cmd ).
21 compile( _, error ) :- synerr( badcommand ).
22
23 tracescan( Cmd ) :- tracescan, !, write( '---scanned'( Cmd ) ), nl.
24 tracescan( _ ).
25 tracecompile( Cmd ) :-
26     tracecompile, !, write( '---compiled'( Cmd ) ), nl.
27 tracecompile( _ ).
28
29 tracescan.          tracecompile.
30
31 % ----- reader and scanner -----
32 % Reader stops on the first dot outside strings.
33 readcmd( String ) :- rdchsk( Ch ), readcmd( Ch, String ).
34
35 readcmd( '.', [] ) :- !, rch.
36 readcmd( '"', ["" | Rest] ) :-
37     !, rdch( Ch ), readstr( Ch, Rest, RestAfter ),
38     rdch( Nextch ), readcmd( Nextch, RestAfter ).
39 readcmd( Ch, [Ch | Rest] ) :- rdch( Nextch ), readcmd( Nextch, Rest ).
40
41 readstr( '"', ["" | Rest], Rest ) :- !.
42 readstr( Ch, [Ch | Rest], RestAfter ) :-
43     rdch( Nextch ), readstr( Nextch, Rest, RestAfter ).
44
45 % This scanner recognizes names, strings, integers, and single
46 % characters. Strings are returned as lists of characters.
47 % The tokens are: n(Name), s(String), i(Integer), ASingleCharacter.
48 tokens( [T | Ts] ) --> token( T ), !, sp, tokens( Ts ).
49 tokens( [] ) --> [].
50
51 token( n( Name ) ) -->
52     letter( L ), namechars( NN ), { pname( Name, [L | NN] ) }.
53 token( s( String ) ) --> [""], stringchars( String ).
54 token( i( Integer ) ) -->
55     sign( S ), digit( D ), digits( DD ),
56     { pname( I, [D | DD] ), signed( S, I, Integer ) }.
57 token( Ch ) --> [Ch].
58
59 letter( Ch ) --> [Ch], { letter( Ch ) }.

```



# **LISTING 8.4 (Continued)**

```

60
61 namechars( [Ch | Chs] ) --> letter( Ch ), !, namechars( Chs ).
62 namechars( [Ch | Chs] ) --> digit( Ch ), !, namechars( Chs ).
63 namechars( [] ) --> [].
64
65 % "" in a string stands for a single ""
66 % (readcmd treats "...""..." as two adjacent strings).
67 stringchars( ["" | Chs] ) --> [ "", "" ], !, stringchars( Chs ).
68 stringchars( [] ) --> [ "" ], !.
69 stringchars( [Ch | Chs] ) --> [Ch], stringchars( Chs ).
70
71 digit( Ch ) --> [Ch], { digit( Ch ) }.
72
73 digits( [D | DD] ) --> digit( D ), !, digits( DD ).
74 digits( [] ) --> [].
75
76 sign( '-' ) --> ['-'].
77 sign( '+' ) --> ['+'].
78 sign( '+' ) --> [].
79
80 signed( '+', !, ! ).
81 signed( '-', !, Integer ) :- Integer is - !.
82
83 sp --> [' '], !, sp. % optional spaces
84 sp --> [].
85
86 % These are used for attributes.
87 qname( Qual-Name ) --> [n( Qual ), '_', n( Name )], !.
88 qname( Variable-Name ) --> [n( Name )]. % ie fits all qualifiers
89
90 constant( Int, integer ) --> [!( Int )], !.
91 constant( Str, string ) --> [s( Str )].
92
93 % ----- symbol table operations -----
94 :- op(100, xfx, ':').
95
96 % Given a relation name and an alias (cf select expression, procedure
97 % rename), use the relation's schema to push a new set of items onto
98 % symbol table stack and to return the generator. The format
99 % of a schema is described with create (procedure newrel).
100 newrelname( RelNm, Alias, Generator, OldST, [Alias:RelST | OldST] ) :-
101     'r e l'( RelNm, Generator, RelST ), !.
102 newrelname( RelNm, _, fail, OldST, OldST ) :- synerr(norelname(RelNm)).
103
104 % Given a qualified name, return its associated variable and type.
105 findattr( Q-Nm, Var, Type, [Q : RelST | ST] ) :-
106     member( attr Nm, Type, Var ), RelST ), !.
107 findattr( QNm, Var, Type, [_ST] ) :- !, findattr(QNm, Var, Type, ST).
108 findattr( QNm, _, _, [] ) :- synerr( noattribute( QNm ) ).
109
110 % ----- command compiler -----
111 % See the various commands for examples of use. Command
112 % interpretation routines listed alongside command grammar processors.
113 command( Cmd ) --> create( Cmd ).
114 command( Cmd ) --> cancel( Cmd ).
115 command( Cmd ) --> select( Cmd ).
116 command( Cmd ) --> relations( Cmd ).
117 command( Cmd ) --> relation( Cmd ).
118 command( Cmd ) --> insert( Cmd ).
119 command( Cmd ) --> delete( Cmd ).

```

# LISTING 8.4 (Continued)

```

120 command( Cmd ) --> update( Cmd ).
121 command( Cmd ) --> stop( Cmd ).
122 command( Cmd ) --> dump( Cmd ).
123 command( Cmd ) --> load( Cmd ).
124
125 % --- create a new relation ---
126 % Eg: create EMP < string name, integer salary, integer dno >.
127 % Eg: create DEPT<integer dno,string manager>.
128 % Eg: create BoardMembers<string name, string pos, integer seniority>.
129 % Note that lower/upper case matters. Keywords must be
130 % in lower case, otherwise use any convention you like.
131 create( newrel( RelName, [V | Vs], [attr(Nm, Type, V) | As] ) ) -->
132     [n( create ), n( RelName )],
133     ['<', typnam( Type, Nm ), typnam( Vs, As ), '>'].
134
135 typnam( [V | Vs], [attr( Nm, Type, V ) | As] ) -->
136     ['.', !, typnam( Type, Nm ), typnam( Vs, As )].
137 typnam( [], [] ) --> [].
138
139 typnam( string, Nm ) --> [n( string), n( Nm )], !.
140 typnam( integer, Nm ) --> [n( integer ), n( Nm )], !.
141 typnam( notype, Nm ) --> synerrc( typeexpected ).
142
143 % A schema stores a pattern for invoking the relation's tuples (the
144 % generator), and a list of symbol table entries linking attribute
145 % names and types with variables in the generator.
146 newrel( RelName, Vars, RelST ) :-
147     not 'r e l'( RelName, _ ), !,
148     mkgen( RelName, Vars, Generator ),
149     assert( 'r e l'( RelName, Generator, RelST ) ).
150 newrel( RelName, _ , _ ) :- namerr( duprelnam( RelName ) ).
151
152 % Add a blank for (rudimentary) security.
153 mkgen( RelName, Vars, Generator ) :-
154     pname( RelName, Chars ), pname( RelNm, [' ' | Chars] ),
155     Generator =.. [RelNm | Vars].
156
157 % --- cancel a relation ---
158 % Eg: cancel EMP.
159 cancel( cancel( RelName ) ) --> [n( cancel ), n( RelName )].
160
161 cancel( RelName ) :- retract( 'r e l'( RelName, Generator, _ ) ), !,
162     retract( Generator ), fail.
163 cancel( RelName ) :- namerr( unknown( RelName ) ).
164
165 % --- queries ---
166 % List the set generated by a select expression.
167 select( ( Generators, Filter, writetuple( Tup ), fail ) ) -->
168     selectexp( set( Generators, Filter, Tup, _ ), [] ).
169
170 writetuple( [] ) :- !, nl.
171 writetuple( [V | Vs] ) :- wri( V ), write( ' ' ), writetuple( Vs ).
172 wri( [X | Y] ) :- !, writetext( [X | Y] ).
173 wri( X ) :- write( X ).
174
175 % List all relations. Eg: relations.
176 relations( ( 'r e l'( RelNm, _ , _ ), write( RelNm ), nl, fail ) ) -->
177     [n( relations )].
178 % List the attributes of a relation. Eg: relation EMP.
179 relation( relation( Name ) ) --> [n( relation ), n( Name )].

```



**LISTING 8.4 (Continued)**

```

180
181 relation( RelN ) :- 'r e l'( RelN, _, Attrs ), !, listattrs( Attrs ).
182 relation( RelN ) :- write( RelN ), write( ' is not a relation!' ), nl.
183
184 listattrs( [] ) :- !.
185 listattrs( [attr( Name, Type, _ ) | Attrs] ) :-
186     write( Type ), write( ' ' ), write( Name ), nl,
187     listattrs( Attrs ).
188
189 % --- select expression ---
190 % Eg: select from EMP, Mgr=EMP, DEPT tuples < name, dno >
191 %     where salary > Mgr_salary*85/100
192 %     and Mgr_name = manager and DEPT_dno = EMP_dno
193 %     and < manager > in (<"Smith">, <"Jones">, <"Brown"> ) .
194 % (ie get names and department numbers of those subordinates of Smith,
195 % Jones or Brown who earn more than 85% of their manager's salary ).
196
197 % Generators pick up tuples from named relations, Filters pass only
198 % tuples fitting the where-clause, Tuple is instantiated to the passed
199 % tuples (one by one), Types is the tuple's pattern with types instead
200 % of attributes (used for type checking).
201 % The example compiles to :
202 % set((' EMP'(Name, Salary, Dno), ' EMP'(MgrName, MgrSalary, MgrDno),
203 %     ' DEPT'(Dno, MgrName) ),
204 %     (Salary > MgrSalary*85/100, true, true,
205 %     (member(MgrName, <"Smith", "Jones", "Brown">), true)),
206 %     [ Name, Dno ], [ string, integer ] )
207
208 selectexp( set( Generators, Filter, Tuple, Types ), InitST ) -->
209     [n( select ), n( from )], relnames( Generators, InitST, ST ),
210     [n( tuples )], tuplepattern( Tuple, Types, ST ),
211     whereclause( Filter, ST ).
212
213 % One or more relation names, possibly "aliased". Symbol table frag-
214 % ments are stacked in reverse order, so attribute search order will
215 % be that of the from-list (using-list for update) relations.
216 relnames( ( Gen, Gens ), OldST, NewST ) -->
217     relname( Name, Alias ), [''], !, relnames( Gens, OldST, TempST ),
218     { newrelname( Name, Alias, Gen, TempST, NewST ) }.
219 relnames( Gen, OldST, NewST ) --> relname( Name, Alias ),
220     { newrelname( Name, Alias, Gen, OldST, NewST ) }.
221
222 relname( Name, Alias ) --> [n( Alias ), '=', n( Name )], !.
223 relname( Name, Name ) --> [n( Name )].
224
225 % tuplepattern is also invoked by inexpr.
226 tuplepattern( [A | As], [T | Ts], ST ) -->
227     [<], attrpatt( A, T, ST ), attrpatts( As, Ts, ST ), [>].
228
229 attrpatts( [A | As], [T | Ts], ST ) -->
230     [''], !, attrpatt( A, T, ST ), attrpatts( As, Ts, ST ).
231 attrpatts( [], [], _ ) --> [].
232
233 attrpatt( Attribute, Type, _ ) --> constant( Attribute, Type ), !.
234 attrpatt( A, T, ST ) --> qname( QN ), { findattr( QN, A, T, ST ) }.
235
236 whereclause( Filter, ST ) --> [n(where)], !, boolexp( Filter, ST ).
237 whereclause( true, _ ) --> [].
238
239 % --- Boolean expressions ---

```



# LISTING 8.4 (Continued)

```

240 % Eg: salary > Mgr_salary * 85/100
241 % or <name> in select from BoardMembers tuples <name>
242 % Note that embedded select expressions do not modify the symbol table,
243 % whose extensions are visible only in the nested constructs.
244 boolexp( E, ST ) --> bterm( T, ST ), rboolexp( T, E, ST ).
245
246 rboolexp( L, ( L : R ), ST ) --> [n( or )], !, boolexp( R, ST ).
247 rboolexp( E, E, _ ) --> [].
248
249 bterm( T, ST ) --> bfactor( F, ST ), rbterm( F, T, ST ).
250
251 rbterm( L, ( L, R ), ST ) --> [n( and )], !, bterm( R, ST ).
252 rbterm( T, T, _ ) --> [].
253
254 bfactor( not F, ST ) --> [n( 'not' )], !, bfactor( F, ST ).
255 bfactor( E, ST ) --> ['('], !, boolexp( E, ST ), [')'].
256 bfactor( E, ST ) --> inexp( E, ST ), !.
257 bfactor( E, ST ) --> relexp( E, ST ).
258
259 % --- set membership ---
260 % Eg: <dno, name > in <1, "Jones">, <2, "Smith">
261 % Eg: <name > in (select from BoardMembers tuples <name>)
262 inexp( ( Generator, Filter ), ST ) -->
263     tuplepattern( Patt, Type, ST ), [n( in )],
264     setexp( set( Generator, Filter, Tuple, Types ), ST ),
265     matchpatterns( Patt, Type, Tuple, Types ).
266
267 % matchpatterns is a rule, so that synerrc can show context.
268 matchpatterns( Patt, Types, Patt, Types ) --> !.
269 matchpatterns( P1, T1, P2, T2 ) -->
270     synerrc( badinexpattern( T1, P1, T2, P2 ) ).
271
272 % --- set expressions ---
273 % A sequence of tuples or a select expression, possibly in parentheses.
274 % The generator for a sequence of tuples is a call on member with the
275 % second parameter instantiated to a list of these tuples.
276 setexp( Set, ST ) --> ['('], !, setexp( Set, ST ), [')'].
277 setexp( Set, ST ) --> selectexp( Set, ST ), !.
278 setexp( set( member( Patt, [Tup|Tups] ), true, Patt, Types ), ST ) -->
279     tuple( Tup, Types ), tuples( Tups, Types ),
280     { mkpattern( Types, Patt ) }, !.
281 setexp( set( fail, fail, [], _ ), _ ) --> synerrc( badsetexpr ).
282
283 tuples( [Tup | Tups], Types ) --> ['('], !, tuple( Tup, TupTypes ),
284     { checktype( Types, TupTypes ) }, tuples( Tups, Types ).
285 tuples( [], _ ) --> [].
286
287 tuple( [A | As], [T | Ts] ) -->
288     ['<'], constant( A, T ), constants( As, Ts ), ['>'], !.
289 tuple( [], _ ) --> ['<'], synerrc( badtuple ), { fail }.
290
291 constants( [A | As], [T | Ts] ) -->
292     ['('], !, constant( A, T ), constants( As, Ts ).
293 constants( [], _ ) --> [].
294
295 checktype( Type, Type ) :- !.
296 checktype( T1, T2 ) :- synerr( inconsistent( T1, T2 ) ).
297
298 % Patt in set( _, _, Patt, _ ) is a list of n fresh variables
299 % (n is the length of tuples in this set).
```



# LISTING 8.4 (Continued)

```

300 mkpattern( [], [] ) :- !.
301 mkpattern( [ _ | Types ], [ V | Vs ] ) :- mkpattern( Types, Vs ).
302
303 % --- relational expressions ---
304 relexp( E, ST ) -->
305     simplex( LeftE, LeftType, ST ), relop( Op ), !,
306     simplex( RightE, RightType, ST ),
307     { consrel( LeftE, LeftType, Op, RightE, RightType, E ) }.
308
309 relop( '=' ) --> ['=', '<'].      relop( '!=' ) --> ['='].
310 relop( '==' ) --> ['<', '>'].    relop( '<' ) --> ['<'].
311 relop( '>=' ) --> ['>', '='].    relop( '>' ) --> ['>'].
312
313 consrel( L, Type, Op, R, Type, E ) :- consrel( L, Op, R, Type, E ), !.
314 consrel( L, LType, Op, R, RType, fail ) :-
315     E =.. [Op, L, R], synerrc( typeconflict( LType, RType, E ) ).
316
317 % The first clause does compile-time equality.
318 consrel( Arg, '==', Arg, _, true ).
319 consrel( L, '==', R, string, fail ).
320 consrel( L, '==', R, string, not L = R ).
321 consrel( L, Op, R, integer, E ) :- E =.. [Op, L, R].
322 consrel( L, '<', R, string, istr( L, R ) ).
323 consrel( L, '==', R, string, ( istr( L, R ) ; L = R ) ).
324 consrel( L, '>', R, string, istr( R, L ) ).
325 consrel( L, '>=', R, string, ( istr( R, L ) ; R = L ) ).
326
327 % Compare strings lexicographically.
328 istr( [], [ _ ] ) :- !.
329 istr( [Ch1 | _], [Ch2 | _] ) :- Ch1 @< Ch2, !.
330 istr( [Ch1 | Chs1], [Ch1 | Chs2] ) :- istr( Chs1, Chs2 ).
331
332 % --- simple expressions ---
333 simplex( E, string, ST ) --> stringexp( E, ST ), !.
334 simplex( E, integer, ST ) --> arithexp( E, ST ).
335
336 stringexp( Str, _ ) --> [s( Str )], !.
337 % Type checking delayed to avoid error messages - might be integer.
338 stringexp( Var, ST ) -->
339     qname( QN ), { findattr( QN, Var, Type, ST ), Type = string }.
340
341 arithexp( E, ST ) --> aterm( T, ST ), rarithexp( T, E, ST ).
342
343 rarithexp( L, E, ST ) -->
344     ['+'], !, aterm( T, ST ), rarithexp( L+T, E, ST ).
345 rarithexp( L, E, ST ) -->
346     ['-'], !, aterm( T, ST ), rarithexp( L-T, E, ST ).
347 rarithexp( E, E, _ ) --> [].
348
349 aterm( T, ST ) --> afactor( F, ST ), raterm( F, T, ST ).
350
351 raterm( L, T, ST ) -->
352     ['*'], !, afactor( F, ST ), raterm( L*F, T, ST ).
353 raterm( L, T, ST ) -->
354     ['/'], !, afactor( F, ST ), raterm( L/F, T, ST ).
355 raterm( T, T, _ ) --> [].
356
357 afactor( E, ST ) --> ['('], !, arithexp( E, ST ), [')'].
358 afactor( Int, _ ) --> [( Int )], !.
359 afactor( Var, ST ) -->

```



# LISTING 8.4 (Continued)

```

360     QName( QN ), { findattr( QN, Var, Type, ST ), Type = integer }, !.
361 afactor( 0, _ ) --> QName( QN ), !, synerrc( notinteger( QN ) ).
362 afactor( 0, _ ) --> synerrc( nointegerfactor ).
363
364 % --- insert ---
365 % Eg: into EMP insert <"Jones",1000,1>, <"Smith",1200,2>.
366 % Eg: into EMP insert select from DEPT tuples <manager, 1050, dno>.
367 insert( ( Generators, Filter, assertz( NewTuple ), fail ) ) -->
368     [n( into ), n( RelName )],
369     { 'r e l'( RelName, _ , RelST ) }, !, [n( insert )],
370     setexp( set( Generators, Filter, Tuple, Types ), [] ),
371     { checktypes( Types, RelST ),
372       mkgen( RelName, Tuple, NewTuple ) }.
373 insert( fail ) --> [n( into ), n( RelNm )],
374     synerrc( norelname( RelNm ) ).
375
376 checktypes( [], [] ) :- !.
377 checktypes( [T|Ts], [attr( _, T, _ )|As] ) :- !, checktypes( Ts, As ).
378 checktypes( Types, Attrs ) :- synerr( badsettype( Types, Attrs ) ).
379
380 % --- delete ---
381 % Eg: from EMP delete all tuples.
382 % Eg: from EMP delete tuples where salary < 1000 and
383 %     <dno> in select from DEPT tuples <dno>
384 %     where manager = "Smith".
385 % (ie fire all subordinates of Smith who earn less than a 1000 )
386 delete( ( RelGen, RelFilter, retract( RelGen ), fail ) ) -->
387     [n( from ), n( RelNm )],
388     { newrelname( RelNm, RelNm, RelGen, [], ST ) },
389     [n( delete )], delfilter( RelFilter, ST ).
390
391 delfilter( true, _ ) --> [n( all ), n( tuples )], !.
392 delfilter( RelFilter, ST ) -->
393     [n( tuples ), n( where )], boolexp( RelFilter, ST ).
394
395 % --- update ---
396 % Eg: update EMP using DEPT, Mgr=EMP
397 %     so that salary = salary + (Mgr_salary - salary)/5
398 %     where salary < Mgr_salary - 1000 and Mgr_name = manager
399 %     and DEPT_dno = dno
400 %     and not <Mgr_name> in
401 %     select from BoardMembers tuples <name>.
402 % (ie to all employees who earn over a 1000 less than their manager
403 % give a raise equal to 20% of the difference, provided the manager
404 % does not sit on the board)
405 % This is compiled to :
406 % ' EMP'( Name, Sal, Dno ), % OldTup
407 % (' DEPT'(Dno,Manager), ' EMP'(Manager,MgrSal,MgrDno)), % UseGens
408 % (Sal < MgrSal - 1000, true, true,
409 %     not (' BoardMembers'(Manager,_,_), true) ), % Filter
410 % NewSal is Sal + (MgrSal - Sal)/5, % Modifications
411 % retract(' EMP'(Name,Sal,Dno)),assert(' EMP'(Name,NewSal,Dno)),fail
412 update( ( OldTup, UseGens, Filter, Modifications,
413     retract( OldTup ), assert( NewTup ), fail ) ) -->
414     [n( update ), n( RelNm )],
415     { 'r e l'( RelNm, OldTup, OldST ),
416       'r e l'( RelNm, NewTup, NewST ), !,
417       makemodlist( OldST, NewST, MList ) },
418     usingclause( UseGens, UseST ), { ST = [RelNm : OldST | UseST] },
419     [n( so ), n( that )],

```



# LISTING 8.4 (Continued)

```

420     modifier( Modification, MList, ST ),
421     modifiers( Modification, Modifications, MList, ST ),
422     { closemodlist( MList ) }, whereclause( Filter, ST ).
423 update( fail ) --> [n( update )], synerr( noupdatedrelation ).
424
425 usingclause( Gens, ST ) --> [n( using )], relnames( Gens, [], ST ).
426 usingclause( true, ST ) --> [].
427
428 modifiers( M, ( M , Ms ), MList, ST ) -->
429     ['.', !, modifier( MM, MList, ST ),
430     modifiers( MM, Ms, MList, ST )].
431 modifiers( M, M, _ ) --> [].
432
433 modifier( AttrVar is Expr, MList, ST ) -->
434     [n( Nm )], { findmname( Nm, AttrVar, Type, MList ) },
435     ['='], simplexp( Expr, EType, ST ),
436     { mtype( Type, EType, Nm ) }.
437
438 % A "modlist" lists updated relation's attributes together with new
439 % variables forming new tuple's attributes and Mod variables which are
440 % used to flag an attribute's modification when it is detected on the
441 % left hand side of an equality in "so that"-list.
442 makemodlist( [Old | Olds], [attr( _ , _ , NewV ) | NewVs],
443     [modif( Old, NewV, Mod ) | Mods] ) :-
444     !, makemodlist( Olds, NewVs, Mods ).
445 makemodlist( [], [], [] ).
446
447 % Bind old and new variables in "modlist" entries with clear flag.
448 closemodlist( [Mod | Mods] ) :-
449     closemod( Mod ), !, closemodlist( Mods ).
450 closemodlist( [] ).
451 closemod( modif( attr( _ , _ , OldV ), OldV, Mod ) ) :- var( Mod ).
452 closemod( _ ).
453
454 % Flag an updated attribute in "modlist".
455 findmname( Nm, NewV, T, MList ) :-
456     member( modif( attr( Nm, T, _ ), NewV, Mod ), MList ), !,
457     mmod( Mod, Nm ).
458 findmname( Nm, _ , _ ) :- synerr( notinupdatedrel( Nm ) ).
459
460 % If no errors, the first clause of mmod fails, the second binds.
461 mmod( Mod, Nm ) :- not var( Mod ), !, synerr( updatedtwice( Nm ) ).
462 mmod( true, _ ).
463
464 mtype( Type, Type, _ ) :- !.
465 mtype( T1, T2, Nm ) :- synerr( typeconflict( T1, Nm, T2 ) ).
466
467 % - - - control commands - - -
468 stop( sequelstop ) --> [n( stop )].
469
470 sequelstop.      % do nothing (cf the main procedure)
471
472 load( consult( FileName ) ) --> [n( load ), n( from ), n( FileName )].
473
474 dump( dump( FileName ) ) --> [n( dump ), n( to ), n( FileName )].
475
476 dump( FileName ) :- tell( FileName ),
477     'r e l'( Nm, Gen, ST ), wclause( 'r e l'( Nm, Gen, ST ) ),
478     Gen, wclause( Gen ), fail.
479 dump( _ ) :- write( 'end.' ), nl, told .

```



#### LISTING 8.4 (Continued)

---

```
480
481 wclause( Cl ) :- writeq( Cl ), write( '.' ), nl.
482
483 % - - - - - error handling ("bare bones" version) - - - - -
484 synerr( Info ) :- synmes( Info ), ancestor( getcommand( _, error ) ).
485
486 synerrc( Info ) --> { synmes( Info ), write( 'Context: ' ) },
487                    context.      % will fail eventually!
488 synerrc( _ ) --> { nl, ancestor( getcommand( _, error ) ) }.
489
490 synmes(Info) :- nl, write('--- Syntactic error: '), write(Info), nl.
491
492 context --> [Token], { wtoken( Token ) }, context.
493
494 wtoken( T ) :- wt( T, RealT ), write( RealT ), write( ' ' ), !.
495 wt( n( Name ), Name ).
496 wt( i( Integer ), Integer ).
497 wt( s( String ), String ).
498 wt( Char, Char ).
499
500 namerr( Info ) :- nl, write( '*** Error: ' ), nl,
501                write( Info ), nl, tagfail( docommand( _, _ ) ).
```



---

## 9 PROLOG DIALECTS<sup>1</sup>

---

### 9.1. PROLOG I

The idea of logic programming emerged in Marseilles in the first half of 1972 while Robert Kowalski was visiting the artificial intelligence team founded by Alain Colmerauer at the University of Marseilles. Colmerauer with his team prepared the design specification of the programming language Prolog (Colmerauer *et al.* 1972). The language resembled a theorem prover rather closely, but it already possessed the essential properties of contemporary Prolog, and even some features reintroduced quite recently, e.g. delaying calls till appropriate instantiation of their arguments. Almost at the same time Kowalski advocated predicate calculus as a formalism for expressing algorithms without commitment to a specific strategy of their execution; the short note (1972) was later expanded to a larger paper (1974). Hence the two pioneers of logic programming took from the outset different approaches to the problem of changing the idea into reality. Although developed in close interaction, these different attitudes still manifest themselves in logic programming research.

The language described in Colmerauer *et al.* (1972) was implemented in Algol W on IBM 360/67 by Philippe Roussel and used at once in several applications (Colmerauer *et al.* 1973, Pasero 1973, Kanoui 1973, Joubert 1974; Bergman and Kanoui 1973, 1975, Battani and Méloni 1975, Guizol 1975). It was quickly replaced by an improved version, coded partly in Fortran by Battani and Méloni (1973) and partly in Prolog by Colmerauer and Roussel. This was the first version of Prolog used outside Marseilles. Although not christened so by its authors, it deserves the name of Prolog I, especially as its commonly used name "Marseille Prolog" has become ambiguous.

<sup>1</sup> This chapter was contributed by Janusz S. Bień, Institute of Informatics, Warsaw University, Warsaw, Poland.



The original reference to Prolog I is Roussel (1975); some historical information can be found in Battani and Méloni (1973) and Kluźniak (1984). The syntax of the language is illustrated below by the sample clauses:

```
+APPEND( NIL, *X, *X ).
+APPEND( *X.*Y, *Z, *X.*V ) -APPEND( *Y, *Z, *V ).
```

This should be preceded by the declaration of the infix dot:

```
-AJOP( "`.`", 1, "X'( X'X )" )!
```

And a sample directive (SORT means *write*):

```
-APPEND( A.B.NIL, C.NIL, *X ) -SORT( *X ) -LIGNE!
```

Positive literals (see Chapter 2) are preceded with +, negative with -; the notation allows representation of non-Horn clauses. This was a natural requirement, because the early versions of Prolog were intended to implement a general theorem-proving method known as linear resolution with selection function (Kowalski and Kuehner 1971). Program clauses were distinguished from directives by a different terminator. The syntax survived its original motivation and is still used in some versions of the language (Kluźniak and Szpakowicz 1983).

## 9.2. PROLOG II

After Prolog I was released, Colmerauer's team experimented with various mutations of the language; some of them have been described in Guizol and Méloni (1976), Colmerauer *et al.* (1979) and Kanoui and Van Caneghem (1980). Finally it was announced that the goal of creating "the ultimate Prolog" was achieved (Colmerauer *et al.* 1981). The new language was called Prolog II by its authors (Colmerauer 1982, Van Caneghem 1982, Kanoui 1982).

The most important innovation of Prolog II is the treatment of cyclic data structures (see Section 1.2.3). They are simply valid representations of infinite trees, which can be manipulated in a similar way to other terms (Colmerauer 1979, 1982). However, the same infinite tree can be represented by different data structures; to let them be matched correctly it appeared necessary to treat functors in the same way as arguments (Filgueiras 1982). As a result, a functor can be a variable or a compound term. In Prolog II, the standard form of terms is considered just a shorthand notation for a more general form called tuple. For example, `ff(x)` stands



for  $\langle \text{ff}, x \rangle$ , while  $\langle x, y \rangle$  and  $\langle \langle \text{ff}(x) \rangle, y \rangle$  are also legal terms (single-letter names denote variables,  $\text{ff}$  is a constant). Instead of unifying two tuples, Prolog II constructs a system of equations. For example, matching  $\langle x \rangle$  with  $\langle \text{ff}(x) \rangle$  corresponds to solving in  $x$  the equation

$$x = \text{ff}(x).$$

The solution of this equation is the infinite tree  $\text{ff}(\text{ff}(\text{ff}(\dots)))$ , which is represented by an appropriate cyclic data structure.

The behavior of Prolog programs is described as incremental solving of the system of equations introduced by program clauses, which can also be seen as rewriting rules. The execution of a call is viewed as the operation of erasing it by applying the rules and solving the appropriate equations. This viewpoint manifests itself in the syntax of clauses by an arrow leading from the head to the (possibly empty) body, e.g.

```
append( nil, x, x ) → ;
append( x.y, z, x.v ) → append( y, z, v ) ;
```

This approach makes it possible to describe the principles of Prolog II in a compact and self-contained way, relieved from the references to theorem-proving techniques and relying only on the most fundamental and intuitive notions of logic (Colmerauer 1983).

Prolog II offers a simple yet powerful coroutining mechanism (see Chapter 2). A call may require its parameter to be instantiated. Given a variable, it waits for it to become bound. This is achieved by the built-in procedure *geler* ("freeze"), whose first argument is a "trigger" (usually a variable to be bound) and the second a call (to be delayed). If the "trigger" is already bound, *geler* simply executes the call.

Another coroutining primitive is *dif*, which succeeds if its parameters are not "perfectly" equal (see the built-in procedure  $=$ , Section 5.6). If during the execution of *dif* ("down the terms' structure") a variable is encountered, *dif* waits until it becomes bound and only then resumes the comparison.

The coroutining mechanism will be illustrated by two examples adapted from Colmerauer *et al.* (1983). The first example is a procedure that takes two trees represented as deeply nested dotted list structures. It succeeds when both structures can be flattened to the same linear list. A built-in procedure *ident* is used to check whether the argument is a constant.

```
sameleaves( a, b ) → leaves( a, u ) leaves( b, u ) list( u );
leaves( a, u ) → geler( u, leaves1( a, u ) );
```



```

leaves1( a, a.nil ) → ident( a );
leaves1( a.l, a.u ) → ident( a ) leaves( l, u );
leaves1( ( a.b ).l, u ) → leaves1( a.b.l, u );

list( nil ) →;
list( a.u ) → list( u );

```

The second example is a procedure that generates a list of digits 1, 2, 3 such that all three elements are different.

```

perm( x.y.z.nil ) → alldifferent( x.y.z.nil )
                    alldigits( x.y.z.nil );

alldigits( nil ) →;
alldigits( x.l ) → digit( x ) alldigits( l );
digit( 1 ) →; digit( 2 ) →; digit( 3 ) →;

alldifferent( nil ) →;
alldifferent( x.l ) → outside( x, l ) alldifferent( l );

outside( x, nil ) →;
outside( x, y.l ) → dif( x, y ) outside( x, l );

```

Prolog II supports a kind of modularisation implemented by so-called “worlds”, each with a unique name. Worlds are organised into a tree structure. The root is the world “origine”, which has two subworlds “ordinaire” and “?????”. Subworlds of “ordinaire” (which is the default) can be created by the user who can walk up and down the tree, and also create and discard worlds. “?????” contains the Prolog II supervisor and cannot be used as the current world. Every procedure name is associated with the world in which it was first mentioned (“declared”). It is accessible in this world and its descendants but not in its siblings. Moreover, a name *N* and the same name *N* declared in a superworld later on refer to different procedures.

Clauses are available for all manipulations (including initial definition) only in the world where the procedure name has been declared and in its direct subworlds. For example, standard procedure names are introduced in “origine”—with clauses defined in “?????”—and used in “ordinaire”. So, the user cannot change a standard procedure definition. Clause indexing is provided, or rather tuple indexing. The leftmost name in the tuple is used as a key.

The purpose of Toy’s *tag*, *tagexit* etc. (see Section 5.12) is served in Prolog II by a pair of built-in procedures *bloc*, *fin-bloc*. In the call *bloc(l, t)*, *t* is the call to be executed in the block. When, during the execution of *t*, a call of the form *fin-bloc(l1)* is encountered, the most recent call on



$bloc(l, t)$  with  $l$  unifiable with  $ll$  is sought. If none is found, an error condition is raised, else this call on  $bloc$  succeeds deterministically. This feature is used for error handling and for exiting loops.

We have presented here only some of the Prolog II features, and the interested reader is referred to Colmerauer *et al.* (1983). It is interesting that the pilot implementation of Prolog II which is described here was done on an Apple II microcomputer using software paging on floppy disks.

### 9.3. MICRO-PROLOG AND MPROLOG

micro-Prolog is the dialect used by Kowalski's team at Imperial College of Science and Technology in London. micro-Prolog was developed and implemented by McCabe (1981); his main goal was to install Prolog on a cheap 8-bit microcomputer. micro-Prolog uses lists to represent terms, calls and clauses, e.g.

```
(( Append() x x ))
(( Append ( x | X ) Y ( x | Z ) ) ( Append X Y Z ) )
```

The list notation has several advantages: predicates and functors need not have a fixed number of arguments (i.e. the whole argument list can be bound to a single variable); their names can be arbitrary list structures (for technical reasons this is not allowed for predicates). micro-Prolog also supports a simple form of modularity: modules are created dynamically and the accessibility of names is determined by export/import lists.

A typical user is not expected to interact directly with micro-Prolog, because a special front-end called Simple is provided to conceal the low-level language features. Here is the *append* example in the Simple syntax:

```
Append( () x x )
Append( ( x | X ) Y ( x | Z ) ) if Append( X Y Z )
```

Simple was used as a computer language for children; other interesting applications are expert systems. The language is subject to various experiments and extensions, such as an explanation facility or an original form of input/output operations called query-the-user (Sergot 1982); Simple and other extensions are all written in micro-Prolog. Both micro-Prolog and some of its applications are extensively documented in Ennals (1983) and Clark and McCabe (1984).

MPROLOG was developed at the Institute for Co-ordination of Computer Techniques, in Budapest (Bendl *et al.* 1980, SzKI 1982), using the programming language CDL2 (Koster 1974). MPROLOG is an upward-



compatible extension of Prolog-10, intended for creating production software for mainframe computers. The crucial extension consists in introducing a form of modularity, based on the ideas of Szeredi (1982) and similar in spirit to that found in many other languages. The modules are syntactic units and contain explicit export/import lists determining the visibility (i.e. the accessibility) of names; a visible name can serve as a functor or as a predicate name. When a program is entered, only the main module is loaded and executed; other modules must be loaded explicitly by calling appropriate built-in procedures.

MPROLOG is a large system which includes several components. The pretranslator produces an internal form of a program module; the consolidator links the modules into a program; the interpreter executes it. The program development support system (PDSS) provides a dedicated editor and debugging aids. A compiler and an optimizer are under development. The language offers a multitude of built-in procedures (probably more than any other Prolog system) and interfaces to user-supplied procedures written in CDL2 and Fortran.



## APPENDICES

Appendix A.1	Kernel File .....	256
Appendix A.2	"Bootstrapper" .....	258
Appendix A.3	User Interface and Utilities .....	263
Appendix A.4	Three Useful Programs	
	A Simple Editor .....	284
	A Primitive Tracing Tool.....	287
	A Program Structure Analyser with Analyser Analysed .....	289

## APPENDIX A.1

### Kernel File

```
1 % KERNEL file
2 % standard atoms
3 ';/2 ';/2 'call'/1 'tag'/1
4 '[]'/0 ';/2 'error'/1 'user'/0
5
6 % atoms identifying system routines (keep 'fail' first and 'true' last)
7 'fail'/0 'tag'/1 'call'/1 'l'/0
8 'tagcut'/1 'tagfail'/1 'tagexit'/1 'ancestor'/1
9 'halt'/1 'status'/0
10 'display'/1 'rch'/0 'lastch'/1 'skipbl'/0 'wch'/1
11 'echo'/0 'noecho'/0
12 'see'/1 'seeing'/1 'seen'/0 'tell'/1 'telling'/1 'told'/0
13 'ordchr'/2 'sum'/3 'prod'/4 'less'/2 '@<'/2
14 'smallletter'/1 'bigletter'/1 'letter'/1 'digit'/1 'alphanum'/1
15 'bracket'/1 'solochar'/1 'symch'/1
16 'eqvar'/2 'var'/1
17 'atom'/1 'integer'/1 'nonvarint'/1
18 'functor'/3 'arg'/3 'pname'/2 'pnamei'/2
19 '$proc'/1 '$proclimit'/0 '$procinit'/0
20 'clause'/5 'retract'/3 'abolish'/2 'assert'/3 'redefine'/0
21 'predefined'/2 'protect'/0
22 'nonexistent'/0 'nononexistent'/0
23 'debug'/0 'nodebug'/0
24 'true'/0
25
26 % kernel library
27 error(:0) : nl . display('+++ System call error: ') . display(:0) .
28 nl . fail . []
29 :ordchr(10, :0) . assert(iseoln(:0), [], 0) .
30         assert(nl, wch(:0), [], 0) . [] #
31 '='(:0, :0) : []
32 ','(:0, :1) : call(:0) . call(:1) . []
33 ','(:0, _) : call(:0) . []
34 ','(_, :0) : call(:0) . []
35 not(:0) : call(:0) . '!' . fail . []
36 not(_) : []
37 check(:0) : not(not(:0)) . []
38 'side_effects'(:0) : not(not(:0)) . []
39
40 once(:0) : call(:0) . '!' . []
41
42 '@=<'(:0, :1) : '@<'(:1, :0) . '!' . fail . []
43 '@=<'(_, _) : []
44 '@>'(:0, :1) : '@<'(:1, :0) . []
45 '@>'(:0, :1) : '@=<'(:1, :0) . []
46
47 % - - - - - basic input procedures - - - - -
48 rdchsk(:0) : rch . skipbl . lastch(:0) . []
49 rdch(:0) : rch . lastch(:1) . sch(:1, :0) . []
50 % convert nonprintable characters to blanks
51 sch(:0, :0) : '@<'(' ', :0) . '!' . []
52 sch(:0, ' ') : []
53
```



```

54 repeat : []
55 repeat : repeat . []
56 member(:0, :0..1) : []
57 member(:0, _:1) : member(:0, :1) . []
58
59 proc(:0) : '$procinit' . '$pr'(:0) . []
60 '$pr'(:0) : '$proclimit' . '!' . fail . []
61 '$pr'(:0) : '$proc'(:0) . []
62 '$pr'(:0) : '$pr'(:0) . []
63
64 % b a g o f (preserves order of solutions)
65 bagof(:0, :1, _) : asserta('BAG'('BAG')) . call(:1) .
66   asserta('BAG'(:0)) . fail . []
67 %% 0 Item, 1 Condition,
68 bagof(_, :0) : 'BAG'(:1) . '!' . intobag(:1, [], :0) . []
69 %% 0 Bag, 1 Item,
70 intobag('BAG', :0, :0) : '!' . retract('BAG', 1, 1) . []
71 %% 0 Final_bag,
72 intobag(:0, :1, :2) : retract('BAG', 1, 1) . 'BAG'(:3) . '!' .
73   intobag(:3, :0..1, :2) . []
74 %% 0 Item, 1 This_bag, 2 Final_bag, 3 Next_item,
75
76 % end of file - toyprolog will now read from the terminal
77 : display('Kernel file loaded.') . nl . see(user) . [] #

```



## APPENDIX A.2

### "Bootstrapper"

---

```

1  % % % translator of Prolog-10(mini) into "kernel-prolog" % % %
2  translate(:0, :1) : see(:0) . tell(:1) . program . seen . told .
3  see(user) . tell(user) . display(translated(:0)) . nl . []
4      %% 0 from_file, 1 to_file
5  % -----
6  % main loop
7  program : rch . skpb(:0) . tag(transl(:0)) . isendsym(:0) . '!' . []
8  program : program . []
9  transl('@') : '!' . rch . []
10 transl('%') : comment('%', :0, []) . '!' . puttr(:0) . []
11 transl(:0) : clause(:0, :1, [], :2) . puttr(:1) . putvarnames(:2, 0).[]
12      %% 0 startch, 1 termrepr, 2 sym_tab
13 isendsym('@') : []      % otherwise fail, ie loop
14 % -----
15 % error handling: skip to the nearest dot
16 err(:0, :1) : display("*** error in ") . display(:0) .
17   display(' unexpected "') . display(:1) . lastch(:2) .
18   display(" . text skipped: ") . skip(:2) . nl . tagfail(transl(_)).[]
19      %% 0 proc_name, 1 bad_item, 2 first_skipped_char
20 skip('.') : wch('.') . []
21 skip(:0) : wch(:0) . rch . lastch(:1) . skip(:1) . []
22 % -----
23 % a comment extends till end of line
24 comment(:0, :0..1, :1) : iseoln(:0) . []
25      %% 0 eoln, 1 rest_of_termrepr
26 comment(:0, :0..1, :2) : rch . lastch(:3) . comment(:3, :1, :2) . []
27      %% 0 char, 1 termrepr, 2 rest_of_termrepr, 3 nextchar
28 % -----
29 % read a goal
30 clause(':', '':0, :1, :2) : '!' . ctail(':', :0, '':@':1, :2) . []
31      %% 0 termrepr, 1 rest_of_termrepr, 2 sym_tab
32 % read an assertion/rule
33 clause(:0, :1, :2, :3) : fterm(:0, :4, :1, '':':5, :3) .
34 '!' . ctail(:4, :5, :2, :3) . []
35      %% 0 fterm_firstch, 1 termrepr, 2 rest_of_termrepr,
36      %% 3 sym_tab, 4 ctail_firstch, 5 middletermrepr
37 clause(:0, _, _ , _) : err(clause, :0) . []
38 % -----
39 % clause tail
40 ctail(':', '':':0, :0, _) : '!' . []
41      %% 0 rest_of_termrepr
42 % righthand side of a non-unit clause, or a goal
43 % eoln and blanks inserted to make the output look tidy
44 ctail(':', :4, '':':0, :1, :2) : rdch('-') . '!' . iseoln(:4) .
45 rdchsk(:3) . ctailaux(:3, :0, :1, :2) . []
46      %% 0 termrepr, 1 rest_of_termrepr, 2 sym_tab, 3 calls_firstch,
47      %% 4 eoln
48 ctail(:0, _, _ , _) : err(ctail, :0) . []
49 % get the righthand side of a clause (embedded comments not displaced)
50 ctailaux('%', :0, :1, :2) : comment('%', :0, '':':':5) . '!' .
51 rdchsk(:3) . ctailaux(:3, :5, :1, :2) . []
52      %% 0 termrepr, 1 rest_of_termrepr, 2 sym_tab, 3 rest_firstch,

```



## APPENDIX A.2 (Continued)

```

53      %% 5 middletermrepr
54      ctailaux(:0, :1, :2, :3) : fterm(:0, :4, :1, ''::5, :3) .
55      fterms(:4, :5, :2, :3) . []
56      %% 0 fterm_firstch, 1 termrepr, 2 rest_of_termrepr,
57      %% 3 sym_tab, 4 fterms firstch, 5 middletermrepr
58      % a list of functor-terms (ie calls)
59      fterms(' ', ''::1):0, :0, _ : ' ' . []
60      %% 0 rest_of_termrepr
61      % eoln and blanks - cf ctail/2/
62      fterms(' ', :4, ''::1):0, :1, :2) : ' ' . iseoln(:4) .
63      rdchsk(:3) . ctailaux(:3, :0, :1, :2) . []
64      %% 0 termrepr, 1 rest_of_termrepr, 2 sym_tab, 3 ctail_firstch,
65      %% 4 eoln
66      fterms(:0, _ , _ ) : err(fterms, :0) . []
67      % -----
68      % a functor-term
69      fterm(:0, :1, ''::2, :3, :4) :
70      ident(:0, :5, :2, ''::6) . ' ' . args(:5, :1, :6, :3, :4) . []
71      %% 0 id_firstch, 1 terminator, 2 termrepr, 3 rest_of_termrepr,
72      %% 4 sym_tab, 5 id_terminator, 6 middletermrepr
73      % identifiers: words, l, quoted names, symbols
74      ident(:0, :1, :0, :2, :3) :
75      wordstart(:0) . rdch(:4) . alphanums(:4, :1, :2, :3) . []
76      %% 0 id_firstch, 1 terminator, 2 termrepr,
77      %% 3 rest_of_termrepr, 4 nextch
78      ident(' ' , :0, ''::1, :1) : rch . skpb(:0) . []
79      %% 0 terminator, 1 termrepr
80      ident(' ', :0, :1, :2) : rdch(:3) . qident(:3, :0, :1, :2) . []
81      %% 0 terminator, 1 termrepr, 2 rest_of_termrepr, 3 nextch
82      ident(:0, :1, :0, :2, :3) :
83      symch(:0) . rdch(:4) . symbol(:4, :1, :2, :3) . []
84      %% 0 symb_firstch, 1 terminator, 2 termrepr,
85      %% 3 rest_of_termrepr, 4 nextch
86      % quoted identifiers
87      qident(' ', :0, :1, :2) :
88      rdch(:3) . qidentail(:3, :0, :1, :2) . ' ' . []
89      %% 0 terminator, 1 termrepr, 2 rest_of_termrepr, 3 nextch
90      qident(:0, :1, :0, :2, :3) : rdch(:4) . qident(:4, :1, :2, :3) . []
91      %% 0 char, 1 terminator, 2 termrepr,
92      %% 3 rest_of_termrepr, 4 nextch
93      qidentail(' ', :0, ''::1, :2) :
94      rdch(:3) . qident(:3, :0, :1, :2) . []
95      %% 0 terminator, 1 termrepr, 2 rest_of_termrepr, 3 nextch
96      qidentail(_ , :0, :1, :1) : skpb(:0) . []
97      %% 0 terminator, 1 rest_of_termrepr
98      % words and symbols
99      alphanums(:0, :1, :0, :2, :3) :
100      alphanum(:0) . ' ' . rdch(:4) . alphanums(:4, :1, :2, :3) . []
101      %% 0 an_alphanum, 1 terminator, 2 termrepr,
102      %% 3 rest_of_termrepr, 4 nextch
103      alphanums(_ , :0, :1, :1) : skpb(:0) . []
104      %% 0 terminator, 1 rest_of_termrepr

```



## APPENDIX A.2 (Continued)

```

105 symbol(:0, :1, :0, :2, :3) :
106     symch(:0) . '!' . rdch(:4) . symbol(:4, :1, :2, :3) . []
107     %% 0 a_symbolchar, 1 terminator, 2 termrepr,
108     %% 3 rest_of_termrepr, 4 nextch
109 symbol(_ , :0, :1, :1) : skpb(:0) . []
110     %% 0 terminator, 1 rest_of_termrepr
111 % get argument list: nothing or a sequence of terms in brackets
112 args('!', :0, '!', :1, :2, :3) :
113     '!' . rdchsk(:4) . terms(:4, :1, :2, :3) . rdchsk(:0) . []
114     %% 0 nextch, 1 termrepr, 2 rest_of_termrepr,
115     %% 3 sym_tab, 4 terms_firstch
116 args(:0, :0, :1, :1, _ ) : []
117     %% 0 nextch, 1 rest_of_termrepr
118 % get a sequence of terms
119 terms(:0, :1, :2, :3) : term(:0, :4, :1, :5, inargs, :3) .
120     termstail(:4, :5, :2, :3) . []
121     %% 0 term_firstch, 1 termrepr, 2 rest_of_termrepr, 3 sym_tab,
122     %% 4 terminator, 5 middletermrepr
123 termstail(')', '!', :0, :0, _ ) : '!' . []
124     %% 0 rest_of_termrepr
125 termstail(' ', '!', :0, :1, :2) :
126     '!' . rdchsk(:3) . terms(:3, :0, :1, :2) . []
127     %% 0 middletermrepr, 1 rest_of_termrepr, 2 sym_tab, 3 nextch
128 termstail(:0, _ , _ , _ ) : err(termstail, :0) . []
129 % -----
130 % get a term (context used to force brackets around lists within lists)
131 term(:0, :1, :2, :3, :4, :5) : t(:0, :1, :2, :3, :4, :5) . '!' . []
132     %% 0 firstch, 1 terminator, 2 termrepr,
133     %% 3 rest_of_termrepr, 4 context, 5 sym_tab
134 term(:0, _ , _ , _ , _ ) : err(term, :0) . []
135 t(:0, :1, :2, :3, _ , :4) : variable(:0, :1, :2, :3, :4) . []
136 t(:0, :1, :2, :3, inargs, :4) : list(:0, :1, :2, :3, :4) . []
137 t(:0, :1, '!', :2, :3, inlist, :4) : list(:0, :1, :2, '!', :3, :4) . []
138 % a dirty patch for negative numbers
139 t('-', :0, :1, :2, _ , :3) :
140     rdch(:4) . numberorterm(:4, :0, :1, :2, :3) . []
141     %% 0 terminator, 1 termrepr, 2 rest_of_termrepr,
142     %% 3 sym_tab, 4 nextch
143 t(:0, :1, :2, :3, _ , _ ) : number(:0, :1, :2, :3) . []
144 t(:0, :1, :2, :3, _ , :4) : fterm(:0, :1, :2, :3, :4) . []
145 % -----
146 numberorterm(:0, :1, '-', :2, :3, _ ) :
147     digit(:0) . '!' . number(:0, :1, :2, :3) . []
148     %% 0 nextch, 1 terminator, 2 termrepr, 3 rest_of_termrepr
149 numberorterm(:0, :1, ' ', :2, :3, :4) :
150     symbol(:0, :5, :2, ' ', :6) . args(:5, :1, :6, :3, :4) . []
151     %% 0 nextch, 1 terminator, 2 termrepr, 3 rest_of_termrepr,
152     %% 4 sym_tab, 5 symbol_terminator, 6 middletermrepr
153 % -----
154 % get a variable
155 variable(:0, :1, :2, :3, :4) : varstart(:0) . alphanums(:0, :1, :5, []).
156     findv(:5, :2, :3, :4) . '!' . []

```



```

157      %% 0 firstch, 1 terminator, 2 termrepr,
158      %% 3 rest_of_termrepr, 4 sym_tab, 5 name
159 findv(' ', :1, :2, :3) : look(:0, 0, :4, :3) . setn(:4, :1, :2).[]
160      %% 0 rest_of_termrepr
161 findv(:0, ' ':1, :2, :3) : look(:0, 0, :4, :3) . setn(:4, :1, :2).[]
162      %% 0 name, 1 termrepr, 2 rest_of_termrepr, 3 sym_tab, 4 num
163 % look counts from 0 and finds the position of a name in the symtab
164 look(:0, :1, :1, :0:2) : []
165      %% 0 name, 1 num, 2 symtabtail
166 look(:0, :2, :1, :0:3) : sum(:2, 1, :4) . look(:0, :4, :1, :3) . []
167      %% 0 name, 1 num, 2 currnum, 3 symtabtail, 4 currnumplus1
168 % set a number: no more than two digits (should be enough)
169 setn(:0, :1:2, :2) : 'less'(:0, 10) .
170     ordchr(:3, '0') . sum(:3, :0, :4) . ordchr(:4, :1) . []
171      %% 0 num, 1 char, 2 rest_of_termrepr, 3 k, 4 kplusnum
172 setn(:0, :1, :2) : 'less'(:0, 100) . prod(10, :3, :4, :0) .
173     setn(:3, :1, :5) . setn(:4, :5, :2) . []
174      %% 0 num, 1 termrepr, 2 rest_of_termrepr,
175      %% 3 numby10, 4 nummod10, 5 middletermrepr
176 setn(:0, :1, :2) : err(setn, :0) . []
177 % -----
178 % get a list in square brackets
179 list(' ', :0, :1, :2, :3) : rdchsk(:4) . endlist(:4, :1, :2, :3) .
180     rdchsk(:0) . []
181      %% 0 terminator, 1 termrepr, 2 rest_of_termrepr,
182      %% 3 sym_tab, 4 nextch
183 endlist(' ', ' ':1, :0, :2) : []
184      %% 0 rest_of_termrepr
185 endlist(:0, :1, :2, :3) :
186     term(:0, :4, :1, ' ':5, inlist, :3) . ltail(:4, :5, :2, :3) . []
187      %% 0 firstch, 1 termrepr, 2 rest_of_termrepr,
188      %% 3 sym_tab, 4 nextch, 5 middletermrepr
189 ltail(' ', ' ':1, :0, :2) : ' ' . []
190      %% 0 rest_of_termrepr
191 ltail(' ', :0, :1, :2) : ' ' . rdchsk(:3) . variable(:3, ' ', :0, :1, :2).[]
192      %% 0 termrepr, 1 rest_of_termrepr, 2 sym_tab, 3 nextch
193 ltail(' ', :0, :1, :2) : ' ' . rdchsk(:3) .
194     term(:3, :4, :0, ' ':5, inlist, :2) . ltail(:4, :5, :1, :2) . []
195      %% 0 termrepr, 1 rest_of_termrepr, 2 sym_tab,
196      %% 3 term firstch, 4 nextch, 5 middletermrepr
197 ltail(:0, :1, :2) : err(ltail, :0) . []
198 % -----
199 % numbers: only natural ones
200 number(:0, :1, :2, :3) : digit(:0) . digits(:0, :1, :2, :3) . []
201      %% 0 firstch, 1 non_digit, 2 termrepr, 3 rest_of_termrepr
202 digits(:0, :1, :0:2, :3) : digit(:0) .
203     ' ' . rdch(:4) . digits(:4, :1, :2, :3) . []
204      %% 0 firstch, 1 non_digit, 2 termrepr, 3 rest_of_termrepr,
205      %% 4 nextch
206 digits(:0, :1, :1) : skpb(:0) . []
207      %% 0 non_digit, 1 rest_of_termrepr
208 % -----

```

```

209 % auxiliary tests
210 wordstart(:0) : smaller(:0) . []
211 varstart(:0) : bigger(:0) . []
212 varstart('_') : []
213 % -----
214 skpb(:0) : skipbl . lastch(:0) . []
215 % -----
216 % output the translation
217 puttr([]) : 'I' . []
218 puttr(:0:1) : wch(:0) . puttr(1) . []
219 putvarnames(:0, _) : var(:0) . 'I' . nl . []
220 %% 0 sym_tab_end
221 putvarnames(:0:1, :2) : nextline(:2) . wch(' ') . display(:2) .
222   puttr(' ':0) . wch(',') . sum(:2, 1, :3) . putvarnames(:1, :3) . []
223 %% 0 curname, 1 sym_tab_tail, 2 currnum, 3 nextnum
224 nextline(:0) : prod(6, _, 0, :0) . 'I' . nl . display(' %') . []
225 %% 0 a_multiple_of_line_size
226 nextline_ : []
227 % % % the end % % %
228 : display("BOOTSTRAPPER" loaded.) . nl . see(user) . [] #

```



## APPENDIX A.3

### User Interface and Utilities

---

```
1 %      Interpreter of Toy-Prolog - the Prolog part.
2 % (c) COPYRIGHT 1983 - Feliks Kluzniak, Stanislaw Szpakowicz
3 %      Institute of Informatics, Warsaw University
4 % .....
5 %      interactive driver - top level
6 % .....
7 ear :- nl, display('Toy-Prolog listening:'), nl, tag(loop).
8 ear :- halt('Toy-Prolog, end of session.').
9
10 loop :- repeat,
11        display('?- '), read(Term, Sym_tab), exec(Term, Sym_tab), fail.
12
13 stop :- tagfail(loop).
14
15 exec('e r r', _) :- !.      % this covers variables, too
16 exec(:-(Goals), _) :- !, once(Goals).
17 exec(N, _) :- integer(N), !, num_clause.
18 exec(Goals, Sym_tab) :-
19     call(Goals, numbervars(Goals, 0, _),
20     printvars(Sym_tab), enough, !.
21 exec(_, _) :- display(no), nl.      % if call(Goals) fails
22
23 enough :- rch, skipbl, lastch(Ch), rch, not(=(Ch, ';')).
24
25 printvars(Sym_tab) :- var(Sym_tab), display(yes), nl, !.
26 printvars(Sym_tab) :- prvars(Sym_tab).
27
28 prvars(Sym_tab) :- var(Sym_tab), !.
29 prvars([var(NameString, Instance) | Sym_tab_tail]) :-
30     writetext(NameString), display(' = '),
31     side_effects(outt(Instance, fd(_, _), q)),
32     % this is equivalent to writeq(Instance) but we avoid
33     % superfluous calls on numbervars - cf WRITE
34     nl, prvars(Sym_tab_tail).
35
36 num_clause :- display('+++ A number can't be a clause.'), nl.
37
38 % read a program upto end. (the only way to define user procedures);
39 % consult/reconsult must be issued from the terminal, and it returns
40 % there ( consult(user) is correct, too )
41 consult(File) :- seeing(OldF), readprog(File), see(OldF).
42 reconsult(File) :-
43     redefine, seeing(OldF), readprog(File), see(OldF), redefine.
44 readprog(user) :- !, getprog.
45 readprog(File) :- see(File), echo, getprog, noecho, seen.
46
47 % the actual job is done by this procedure
48 getprog :- repeat, read(T), assimilate(T), =(T, end), !.
49
50 assimilate('e r r') :- !.      % a variable is erroneous, too
```



---

```

51 assimilate( -->(Left, Right) ) :-
52     !, tag(transl_rule(Left, Right, Clause)), assertz(Clause).
53 assimilate( :- (Goal) ) :- !, once(Goal).
54 assimilate(end) :- !.
55 assimilate(N) :- integer(N), !, num_clause.
56 % otherwise - store the clause
57 assimilate(Clause) :- assertz(Clause).
58
59
60
61 % .....
62 %           reading a term
63 % .....
64 read(T) :- read(T, Sym_tab).
65 read(T, Sym_tab) :-
66     gettr(T_internal, Sym_tab), !, maketerm(T_internal, T).
67 % if gettr fails, then...
68 read('e r r', _) :-
69     nl, display('+++ Bad term on input. Text skipped: '), skip, nl.
70
71 % skip to the nearest full stop not in quotes or in comment
72 skip :- lastch(Ch), wch(Ch), skip(Ch).
73
74 skip(.) :- rch, lastch(Ch), e_skip(Ch), !.
75 skip('%') :- skip_comment, !, rch, skip.
76 skip(Q) :- isquote(Q), skip_s(Q), !, rch, skip.
77 skip(_) :- rch, skip.
78
79 % stop on a "layout" character
80 e_skip(Ch) :- @=<(Ch, ' ').
81 e_skip(Ch) :- wch(Ch), rch, skip.
82
83 skip_comment :- repeat, rch, lastch(Ch), wch(Ch), iseoln(Ch), !.
84
85 isquote(''). isquote('').
86
87 % skip a string
88 skip_s(Q) :- repeat, rch, lastch(Ch), wch(Ch), =(Ch, Q), !.
89
90 % .....
91 %           parser
92 % .....
93 % This is an operator precedence parser for Prolog-10. g e t t r
94 % constructs the internal representation of a term. Next, m a k e
95 % t e r m constructs the term proper - see r e a d. Here is an in-
96 % formal description of the underlying operator precedence grammar
97 % (each "rule" corresponds to one clause of r e d u c e). Sides are
98 % separated by ==> and multiple righthand sides by OR.
99 %   t ==> variable OR integer OR string
100 %   t ==> identifier
101 %   t ==> identifier ( t )
102 %   t ==> [] OR {}

```



## APPENDIX A.3 (Continued)

```

103 %      t ==> (t) OR [t] OR {t}
104 %      t ==> [t|t]
105 %      t ==> t postfix_funcor
106 %      t ==> t infix_funcor t
107 %      t ==> prefix_funcor t
108 % Sequences of terms separated by commas - in rules 3, 5, 6 - will be
109 % recognised as comma-terms (commas are infix functors, covered by
110 % rule 8). There are five types of operators, vns(_), id(_),
111 % ff(_ , _), br(_ , _), bar: see the scanner. The terminal symbol dot
112 % never gets onto the stack. The terminal symbol bottom is never re-
113 % turned by the scanner; it is only used to initiate and terminate the
114 % main loop (p a r s e). The only nonterminal symbol is t(_).
115 % There are five types of internal representations (Args denotes the
116 % representation of arguments - usually a comma-term):
117 %      tr(Name, Args) - for functor-terms,
118 %      arg0(X)        - for X a variable, atom, number, or string,
119 %      bar(X, Y)       - for a list with front X and tail Y,
120 %      tr1(Name, X)    - for prefix and postfix functors,
121 %      tr2(Name, X, Y) - for infix functors.
122 % A Name in tr may be a bracket type. See r e d u c e (clauses 5, 6)
123 % and m a k e t e r m for details.
124
125 % - - - get the internal representation of a term
126 gettr(X, Sym_tab) :-
127     gettoken(T, Sym_tab), parse([bottom], T, X, Sym_tab).
128
129 % p a r s e takes 4 parameters: the current stack, the current token
130 % from input, the variable used to bring the internal representation
131 % to the surface, and the symbol table (used by g e t t o k e n)
132 parse([t(X), bottom], dot, X, _) :- !.
133 parse(Stack, Input, X, Sym_tab) :-
134     topterminal(Stack, Top, Pos),
135     establish_precedence(Top, Input, Pos, Rel, RTop, RInput),
136     exch_top(Top, RTop, Stack, RStack),
137     step(Rel, RInput, RStack, NewStack, NewInput, Sym_tab),
138     parse(NewStack, NewInput, X, Sym_tab).
139
140 % the topmost terminal will be covered by at most one nonterminal
141 % (the third parameter gives Top's position: 1 on the top, 2 covered)
142 topterminal([t(_), Top | _], Top, 2) :- !.
143 topterminal([Top | _], Top, 1).
144
145 % change the topmost terminal (applies only to mixed functors)
146 exch_top(Top, Top, Stack, Stack) :- !.
147 exch_top(_ , RTop, [t(X), _ | S], [t(X), RTop | S]) :- !.
148 exch_top(_ , RTop, [_ | S], [RTop | S]).
149
150 % - - - perform one step: shift (stack the current token) or reduce
151 step(lseq, RInput, Stack, [RInput | Stack], NewInput, Sym_tab) :-
152     !, gettoken(NewInput, Sym_tab).
153 step(gt, RInput, Stack, NewStack, RInput, _) :-
154     reduce(Stack, NewStack), !.

```



### APPENDIX A.3 (Continued)

```

155 % fail if reduction impossible (parse and gettr will fail, too -
156 %   this failure will be intercepted by gettr's caller)
157
158 % reduce top segment of the stack according to the underlying grammar
159 reduce([ vns(X) | S ], [ t(arg0(X)) | S ]).
160 reduce([ id(l) | S ], [ t(arg0(l)) | S ]).
161 reduce([ br(r, '()'), t(X), br(l, '()'), id(l) | S ],
162        [ t(tr(l, X)) | S ]).
163 reduce([ br(r, Type), br(l, Type) | S ],
164        [ t(arg0(Type)) | S ]) :- not(=(Type, '()')).
165        % '[]' or '{}', see p, 2nd clause
166 reduce([ br(r, Type), t(X), br(l, Type) | S ],
167        [ t(tr(Type, X)) | S ]).
168 reduce([ br(r, '[]'), t(Y), bar, t(X), br(l, '[]') | S ],
169        [ t(bar(X, Y)) | S ]).
170 reduce([ ff(l, Type, _), t(X) | S ],
171        [ t(tr1(l, X)) | S ]) :-
172        ismposf(Type).
173 reduce([ t(Y), ff(l, Type, _), t(X) | S ],
174        [ t(tr2(l, X, Y)) | S ]) :-
175        isminf(Type).
176 reduce([ t(X), ff(l, Type, _) | S ],
177        [ t(tr1(l, X)) | S ]) :-
178        ismpref(Type).
179 % otherwise fail (cf s t e p)
180
181 % - - - auxiliary tests for the parser
182 ispref(fy). ispref(fx).
183
184 ispostf(yf). ispostf(xf).
185
186 ismpref([TUn]) :- ispref(TUn).
187 ismpref([_, TUn]) :- ispref(TUn).
188
189 isminf([TBin]) :- member(TBin, [xfy, yfx, xfx]).
190 isminf([_, _]).
191
192 ismposf([TUn]) :- ispostf(TUn).
193 ismposf([_, TUn]) :- ispostf(TUn).
194
195 % - - - establish precedence relation between the topmost
196 % terminal on the stack and the current input terminal
197 establish_precedence(Top, Input, Pos, Rel, RTop, RInput) :-
198     p(Top, Input, Pos, Rel0),
199     finalize(Rel0, Top, Input, Rel, RTop, RInput), !.
200
201 finalize(lseq, Top, Input, lseq, Top, Input).
202 finalize(gt, Top, Input, gt, Top, Input).
203 finalize(lseq(RTop, RInput), _, _, lseq, RTop, RInput).
204 finalize(gt(RTop, RInput), _, _, gt, RTop, RInput).
205
206 p(id(_), br(l, '()'), 1, lseq).

```



### APPENDIX A.3 (Continued)

```

207 p(br(l, Type), br(r, Type), _, lseq).
208 p(br(l, []), bar, 2, lseq).
209 p(bar, br(r, []), 2, lseq).
210
211 p(Top, Input, 1, gt) :-
212     vns_id br(Top, r), br_bar(Input, r).
213 p(Top, ff(N, Types, P), 1, gt(Top, ff(N, RTypes, P))) :-
214     vns_id br(Top, r), restrict(Types, [fx, fy], RTypes).
215 p(Top, Input, 1, lseq) :-
216     br_bar(Top, l), vns_id br(Input, l).
217 p(Top, ff(N, Types, P), Pos, lseq(Top, ff(N, RTypes, P))) :-
218     br_bar(Top, l), pre_inpost(Pos, Types, RTypes).
219 p(ff(N, Types, P), Input, Pos, gt(ff(N, RTypes, P), Input)) :-
220     br_bar(Input, r), post_inpre(Pos, Types, RTypes).
221 p(ff(N, Types, P), Input, 1, lseq(ff(N, RTypes, P), Input)) :-
222     vns_id br(Input, l), restrict(Types, [xf, yf], RTypes).
223
224 % functors with equal priorities
225 p(ff(NTop, TsTop, P), ff(NInp, TsInp, P), Pos, Rel) :-
226     res_confl(TsTop, TsInp, Pos, RTsTop, RTsInp, Rel0),
227     !, do_rel(Rel0, ff(NTop, RTsTop, P), ff(NInp, RTsInp, P), Rel).
228 % different priorities
229 p(ff(NTop, TsTop, PTop), ff(NInp, TsInp, Plnp), Pos,
230     gt(ff(NTop, RTsTop, PTop), ff(NInp, RTsInp, Plnp))) :-
231     stronger(PTop, Plnp), !,
232     restrict(TsInp, [fx, fy], RTsInp),
233     post_inpre(Pos, TsTop, RTsTop).
234 p(ff(NTop, TsTop, PTop), ff(NInp, TsInp, Plnp), Pos,
235     lseq(ff(NTop, RTsTop, PTop), ff(NInp, RTsInp, Plnp))) :-
236     stronger(Plnp, PTop), !,
237     restrict(TsTop, [xf, yf], RTsTop),
238     pre_inpost(Pos, TsInp, RTsInp).
239
240 p(_, dot, _, gt).
241 p(bottom, _, _, lseq).
242 % otherwise fail (p a r s e fails, too)
243
244 vns_id br(vns(_), _).
245 vns_id br(id(_), _).
246 vns_id br(br(LeftRight, _), LeftRight).
247
248 br_bar(br(LeftRight, _), LeftRight).
249 br_bar(bar, _).
250
251 stronger(Prior1, Prior2) :- less(Prior1, Prior2).
252
253 pre_inpost(1, Types, RTypes) :- % the functor must be prefix
254     restrict(Types, [xf, yf], A),
255     restrict(A, [xfy, yfx, xfx], RTypes).
256 pre_inpost(2, Types, RTypes) :- % the functor must not be prefix
257     restrict(Types, [fx, fy], RTypes).
258

```



```

259 post_inpre(1, Types, RTypes) :- % the functor must be postfix
260     restrict(Types, [fx, fy], A),
261     restrict(A, [xly, yfx, xfx], RTypes).
262 post_inpre(2, Types, RTypes) :- % the functor must not be postfix
263     restrict(Types, [xf, yf], RTypes).
264
265 % leave only those types that do not belong to RSet,
266 % fail if this would leave no types at all (RSet
267 % contains only binary types, or only unary types)
268 restrict([T], RSet, [T]) :- !, not(member(T, RSet)).
269 restrict([TBin, TUn], RSet, [TBin]) :- member(TUn, RSet), !.
270 restrict([TBin, TUn], RSet, [TUn]) :- member(TBin, RSet), !.
271 restrict(Types, _, Types).
272
273 % compute relation for two functors with equal priorities; four cases:
274 %   both normal, Top mixed, Input mixed, both mixed
275 res_confl([TTop], [TInp], Pos, [TTop], [TInp], Rel0) :-
276     !, ff_p(TTop, TInp, Pos, Rel0).
277 res_confl([TTopBin, TTopUn], [TInp], Pos, RTsTop, [TInp], Rel0) :-
278     !, ff_p(TTopBin, TInp, Pos, RelB),
279     ff_p(TTopUn, TInp, Pos, RelU),
280     match_rels(RelB, RelU, Rel0, TTopBin, TTopUn, RTsTop).
281 res_confl([TTop], [TInpBin, TInpUn], Pos, [TTop], RTsInp, Rel0) :-
282     !, ff_p(TTop, TInpBin, Pos, RelB),
283     ff_p(TTop, TInpUn, Pos, RelU),
284     match_rels(RelB, RelU, Rel0, TInpBin, TInpUn, RTsInp).
285 res_confl([TTopBin, TTopUn], [TInpBin, TInpUn], Pos, RTsTop, RTsInp,
286     Rel0) :- ff_p(TTopBin, TInpBin, Pos, RelBB),
287     ff_p(TTopBin, TInpUn, Pos, RelBU),
288     ff_p(TTopUn, TInpBin, Pos, RelUB),
289     ff_p(TTopUn, TInpUn, Pos, RelUU),
290     res_mixed(RelBB, RelBU, RelUB, RelUU, Rel0,
291     TTopBin, TTopUn, TInpBin, TInpUn, RTsTop, RTsInp), !.
292
293 do_rel(lseq, TopF, InpF, lseq(TopF, InpF)).
294 do_rel(gt, TopF, InpF, gt(TopF, InpF)).
295 % fail if Rel0 = err
296
297 match_rels(Rel, Rel, Rel, TBin, TUn, [TBin, TUn]) :- !. % err included
298 match_rels(err, Rel, Rel, _, TUn, [TUn]) :- !.
299 match_rels(Rel, err, Rel, TBin, _, [TBin]) :- !.
300 match_rels(_, _, err, TBin, TUn, [TBin, TUn]).
301
302 res_mixed(Rel0, Rel0, Rel0, Rel0, Rel0,
303     TTopBin, TTopUn, TInpBin, TInpUn,
304     [TTopBin, TTopUn], [TInpBin, TInpUn]).
305 res_mixed(err, err, RelUB, RelUU, Rel0,
306     _, TTopUn, TInpBin, TInpUn, [TTopUn], RTsInp) :-
307     match_rels(RelUB, RelUU, Rel0, TInpBin, TInpUn, RTsInp).
308 res_mixed(RelBB, RelBU, err, err, Rel0,
309     TTopBin, _, TInpBin, TInpUn, [TTopBin], RTsInp) :-
310     match_rels(RelBB, RelBU, Rel0, TInpBin, TInpUn, RTsInp).

```



```

311 res_mixed(err, RelBU, err, RelUU, Rel0,
312           TTopBin, TTopUn, _, TInpUn, RTsTop, [TInpUn]) :-
313     match_rels(RelBU, RelUU, Rel0, TTopBin, TTopUn, RTsTop).
314 res_mixed(RelBB, err, RelUB, err, Rel0,
315           TTopBin, TTopUn, TInpBin, _, RTsTop, [TInpBin]) :-
316     match_rels(RelBB, RelUB, Rel0, TTopBin, TTopUn, RTsTop).
317 res_mixed(_, _, _, err, _, _, _, _, _).
318
319 % establish precedence relation for two (basic) types
320 ff_p(TTop, TInp, Pos, lseq) :-
321     member(TTop, [xfy, fy]),           % right_associative
322     ff_p_aux1(Pos, TInp), !.
323 ff_p(TTop, TInp, Pos, gt) :-
324     member(TInp, [yfx, yf]),          % left_associative
325     ff_p_aux2(Pos, TTop), !.
326 ff_p(_, _, _, err).
327
328 ff_p_aux1(1, TInp) :- ispref(TInp).
329 ff_p_aux1(2, TInp) :- member(TInp, [xfy, xf, xfx]).
330
331 ff_p_aux2(1, TTop) :- ispostf(TTop).
332 ff_p_aux2(2, TTop) :- member(TTop, [yfx, fx, xfx]).
333
334 % .....
335 %   internal representation ----> term
336 % .....
337 maketerm(arg0(X), X) :- !.           % variable, atom, number, string
338 maketerm(tr(')', RawTerm), T) :-
339     !, maketerm(RawTerm, T).
340 maketerm(bar(RawList, RawTail), T) :-
341     !, maketerm(RawTail, Tail),
342     makelist(RawList, Tail, T).
343 maketerm(tr('[', RawList), T) :-
344     !, makelist(RawList, '[', T).
345 maketerm(tr('{}', RawArg), '{}'(Arg)) :-
346     !, maketerm(RawArg, Arg).
347 maketerm(tr(Name, RawArgs), T) :-
348     !, makelist(RawArgs, '[', Args),
349     =..(T, [Name | Args]).
350 maketerm(tr2(Name, RawArg1, RawArg2), T) :-
351     !, maketerm(RawArg1, Arg1), maketerm(RawArg2, Arg2),
352     =..(T, [Name, Arg1, Arg2]).
353 maketerm(tr1(Name, RawArg), T) :-
354     maketerm(RawArg, Arg), =..(T, [Name, Arg]).
355
356 % comma-term to dot-list-with-Tail
357 makelist(tr2(' ', RawArg, RawArgs), Tail, [Arg | Args]) :-
358     !, maketerm(RawArg, Arg), makelist(RawArgs, Tail, Args).
359 makelist(RawArg, Tail, [Arg | Tail]) :- maketerm(RawArg, Arg).
360

```



```

361 % .....
362 % scanner
363 % .....
364 % this scanner returns six kinds of tokens:
365 % vns(_) variables, numbers, strings
366 % id(Name) atoms
367 % ff(Name, Types, Prior) "fix" functors
368 % br(Which, Type) brackets (left/right, '()'/'[]'/'{}')
369 % bar | (in lists)
370 % dot . followed by a layout character
371
372 % - - - read a token and construct its internal form
373 % the input is supposed to be positioned
374 % over the first character of a token (or preceding "white space")
375 gettoken(Token, Sym_tab) :-
376     skipbl, lastch(Startch), absorbtoken(Startch, Rawtoken), !,
377     maketoken(Rawtoken, Token, Sym_tab), !.
378
379 % - - - read in a suitable sequence of characters
380 % a word, ie a regular alphanumeric identifier
381 absorbtoken(Ch, id([Ch | Wordtail])) :-
382     wordstart(Ch), getword(Wordtail).
383 % a variable
384 absorbtoken(Ch, var([Ch | Tail])) :-
385     varstart(Ch), getword(Tail).
386 % a solo character is a comma, a semicolon or an exclamation mark
387 absorbtoken(Ch, id([Ch])) :- solochar(Ch), rch.
388 % a bracket, ie ( ) [ ] { }
389 absorbtoken(Ch, br(Wh, Type)) :-
390     bracket(Ch), bracket(Ch, Wh, Type), rch.
391 absorbtoken('!', bar) :- rch.
392 % a string in quotes or in double quotes
393 absorbtoken("", qid(Qname)) :-
394     rdch(Nextch), getstring("", Nextch, Qname).
395 absorbtoken("", str(String)) :-
396     rdch(Nextch), getstring("", Nextch, String).
397 % a positive number
398 absorbtoken(Ch, num([Ch | Digits])) :-
399     digit(Ch), getdigits(Digits).
400 % a negative number or a dash (possibly starting a symbol, see below)
401 absorbtoken(-, Rawtoken) :- rdch(Ch), num_or_sym(Ch, Rawtoken).
402 absorbtoken(., Rawtoken) :- rdch(Ch), dot_or_sym(Ch, Rawtoken).
403 % a symbol, built of . : - < = > + / * ? & $ @ # ^ _ '
404 absorbtoken(Ch, id([Ch | Symbs])) :- symch(Ch), getsym(Symbs).
405 % an embedded comment
406 absorbtoken('%', Rawtoken) :-
407     skipcomment, lastch(Ch), absorbtoken(Ch, Rawtoken).
408 % this shouldn't happen:
409 absorbtoken(Ch, _) :- display(errinscan(Ch)), nl, fail.
410
411 num_or_sym(Ch, num([-, Ch | Digits])) :-
412     digit(Ch), getdigits(Digits).

```



### APPENDIX A.3 (Continued)

```

413 num_or_sym(Ch, id([-, Ch | Symbs])) :- symch(Ch), getsym(Symbs).
414 num_or_sym(_, id([_])).
415
416 % layout characters precede ' ' in ASCII
417 dot_or_sym(Ch, dot) :- @=<(Ch, ' '). % no advance
418 dot_or_sym(Ch, id([_, Ch | Symbs])) :- symch(Ch), getsym(Symbs).
419 dot_or_sym(_, id([_])).
420
421 skipcomment :- lastch(Ch), iseqnl(Ch), skipbl, !.
422 skipcomment :- rch, skipcomment.
423
424 % - - - auxiliary input procedures
425 % read an alphanumeric identifier
426 getword([Ch | Word]) :-
427     rdch(Ch), alphanum(Ch), !, getword(Word).
428 getword([]).
429
430 % read a sequence of digits
431 getdigits([Ch | Digits]) :-
432     rdch(Ch), digit(Ch), !, getdigits(Digits).
433 getdigits([]).
434
435 % read a symbol
436 getsym([Ch | Symbs]) :-
437     rdch(Ch), symch(Ch), !, getsym(Symbs).
438 getsym([]).
439
440 % read a quoted id or string (Delim is either ' or ")
441 getstring(Delim, Delim, Str) :-
442     !, rdch(Nextch), twodelims(Delim, Nextch, Str).
443 getstring(Delim, Ch, [Ch | Str]) :-
444     rdch(Nextch), getstring(Delim, Nextch, Str).
445 twodelims(Delim, Delim, [Delim | Str]) :-
446     !, rdch(Nextch), getstring(Delim, Nextch, Str).
447 twodelims(_, _, []). %close the list
448
449 % - - - auxiliary tests
450 wordstart(Ch) :- smalletter(Ch).
451 varstart(Ch) :- bigletter(Ch).
452 varstart('_').
453 bracket('(', l, '()').    bracket(')', r, '()').
454 bracket('[', l, '[]').    bracket(']', r, '[]').
455 bracket('{', l, '{}').    bracket('}', r, '{}').
456
457 % - - - transform a raw token into its final form
458 maketoken(var(Namestring), vns(Ptr), Sym_tab) :-
459     makeptr(Namestring, Ptr, Sym_tab).
460 maketoken(id(Namestring), Token, _) :-
461     pname(Name, Namestring), make_ff_or_id(Name, Token).
462 maketoken(qid(Namestring), id(Name), _) :-
463     pname(Name, Namestring).
464 maketoken(num([_ | Digits]), vns(N), _) :-

```



```

465      pnamei(N1, Digits), sum(N, N1, 0).
466 maketoken(num(Digits), vns(N), _) :- pnamei(N, Digits).
467 maketoken(str(Chars), vns(Chars), _).
468 maketoken(Token, Token, _) % br(,) and bar and dot
469
470 % variables are kept in a symbol table (an open list)
471 makeptr(['_'], _, _). %no search - an anonymous variable
472 makeptr(Nmstr, Ptr, Sym_tab) :- look_var(var(Nmstr, Ptr), Sym_tab).
473
474 % look-up
475 look_var(Item, [Item | Sym_tab]).
476 look_var(Item, [_ | Sym_tab]) :- look_var(Item, Sym_tab).
477
478 make_ff_or_id(Name, ff(Name, Types, Prior)) :-
479     'FF'(Name, Types, Prior), !.
480 make_ff_or_id(Name, id(Name)).
481
482 % .....
483 %      grammar rule preprocessor
484 % .....
485 transl_rule(Left, Right, Clause) :-
486     two_ok(Left, Right),
487     isolate_lhs_t(Left, Nont, Lhs_t),
488     connect(Lhs_t, Outpar, Finalvar),
489     expand(Nont, Initvar, Outpar, Head),
490     makebody(Right, Initvar, Finalvar, Body, Alt_flag),
491     do_clause(Body, Head, Clause).
492
493 do_clause(true, Head, Head) :- !.
494 do_clause(Body, Head, _:(Head, Body)).
495
496 % Lhs_t is a list (possibly empty) of lefthand side terminals
497 isolate_lhs_t(' ', Nont, Lhs_t, Nont, Lhs_t) :-
498     ' '(nonvarint(Nont), rulererror(varint)),
499     ' '(isclosedlist(Lhs_t), rulererror(ter)), !.
500 isolate_lhs_t(Nont, Nont, []).
501
502 % fail if not a closed list
503 isclosedlist(L) :- check(isccl(L)).
504 isccl(L) :- var(L), !, fail.
505 isccl([]).
506 isccl([_ | L]) :- isccl(L).
507
508 % connect terminals to the nearest nonterminal's input parameter
509 % (actually, "open" a closed list)
510 connect([], Nextvar, Nextvar) :- !.
511 connect([Tsymb | Tsyms], [Tsymb | Outpar], Nextvar) :-
512     connect(Tsyms, Outpar, Nextvar).
513
514 % - - - translate the righthand side (loop over alternatives)
515 % in alternatives, each righthand side is preceded by a dummy
516 % nonterminal, as defined by 'dummy' --> []. (since terminals

```



```

517 % are appended to input parameters, the input parameter of a common
518 % lefthand side must be a variable)
519 makebody(':(Alt, Alts), Initvar, Finalvar,
520         ':((' dummy'(Initvar, Nextvar), Alt_b), Alt_bs), _) :-
521     !, two_ok(Alt, Alts),
522     makeright(Alt, Nextvar, Finalvar, Alt_b),
523     makebody(Alts, Initvar, Finalvar, Alt_bs, alt).
524 makebody(Right, Initvar, Finalvar, Body, Alt_flag) :-
525     var(Alt_flag), !, % only one alternative
526     makeright(Right, Initvar, Finalvar, Body).
527 makebody(Right, Initvar, Finalvar,
528         ':((' dummy'(Initvar, Nextvar), Body), alt) :-
529     makeright(Right, Nextvar, Finalvar, Body).
530
531 % - - - translate one alternative
532 makeright(':(Item, Items), Thispar, Finalvar, T_item_items) :-
533     !, two_ok(Item, Items),
534     transl_item(Item, Thispar, Nextvar, T_item),
535     makeright(Items, Nextvar, Finalvar, T_items),
536     combine(T_item, T_items, T_item_items).
537 makeright(Item, Thispar, Finalvar, T_item) :-
538     transl_item(Item, Thispar, Finalvar, T_item).
539
540 combine(true, T_items, T_items) :- !.
541 combine(T_item, true, T_item) :- !.
542 combine(T_item, T_items, ':(T_item, T_items)).
543
544 % - - - translate one item (sure to be a functor-term)
545 transl_item(Terminals, Thispar, Nextvar, true) :-
546     !, isclosedlist(Terminals),
547     !, connect(Terminals, Thispar, Nextvar).
548 % conditions (the cut and others)
549 transl_item(!, Thispar, Thispar, !) :- !.
550 transl_item('{}'(Cond), Thispar, Thispar, call(Cond)) :- !.
551 % bad list of terminals (missed the first clause)
552 transl_item([_ | _], _, _, _) :- rulererror(ter).
553 % a nested alternative
554 transl_item(':(X, Y), Thispar, Nextvar, Transl) :-
555     !, makebody(':(X, Y), Thispar, Nextvar, Transl, _).
556 % finally, a regular nonterminal
557 transl_item(Nont, Thispar, Nextvar, Transl) :-
558     expand(Nont, Thispar, Nextvar, Transl).
559
560 % add input parameter and output parameter
561 expand(Nont, In_par, Out_par, Call) :-
562     =..(Nont, [Fun | Args]),
563     =..(Call, [Fun, In_par, Out_par | Args]).
564
565 % - - - error handling
566 two_ok(X, Y) :- nonvarint(X), nonvarint(Y), !.
567 two_ok(_, _) :- rulererror(varint).
568

```



# • APPENDIX A.3 (Continued)

```

569 rulerror(Message) :-
570     nl, display('+++ Error in this rule: ', mes(Message), nl,
571     tagfail(transl_rule(_, _))).
572 % diagnostics are only very brief (and not too informative ...)
573 mes(varint) :- display('variable or integer item.').
574 mes(ter) :- display('terminals not on a closed list.').
575
576 % - - - initiate grammar processing
577 phrase(Nont, Terminals) :-
578     nonvarint(Nont), !,
579     expand(Nont, Terminals, [], Init_call),
580     call(Init_call).
581 phrase(N, T) :- error(phrase(N, T)).
582
583 'dummy'(X, X).
584
585 % *****
586 % *****
587 %   l i b r a r y
588 % *****
589 % *****
590 % *****
591 %   =.. (read as "univ")
592 % *****
593 =..(X, Y) :- var(X), var(Y), !, error(=..(X, Y)).
594 =..(Num, [Num]) :- integer(Num), !.
595 =..(Term, [Fun | Args]) :-
596     setarity(Term, Args, N),
597     functor(Term, Fun, N),          % this works both ways
598     not(integer(Fun)),              % we don't want eg 17(X)
599     setargs(Term, Args, 0, N).      % this works both ways, too
600
601 setarity(Term, Args, N) :- var(Term), !, length(Args, N).
602 % notice that bad Args give an error in l e n g t h
603 setarity(_, _, _). % Arity will be set by f u n c t o r in =..
604
605 % both numeric parameters are given,
606 % the loop stops when the third reaches the fourth
607 % (works both ways because a r g does)
608 setargs(_, [], N, N) :- !.
609 setargs(Term, [Arg | Args], K, N) :-
610     sum(K, 1, K1), arg(K1, Term, Arg),
611     setargs(Term, Args, K1, N).
612
613 % find the length of a closed list; error if not closed
614 length(List, N) :- length(List, 0, N).
615
616 % this is a tail-recursive formulation of length
617 length(L, _, _) :- var(L), !, error(length(L, _)).
618 length([], N, N) :- !.
619 length([_ | List], K, N) :-
620     !, sum(K, 1, K1), length(List, K1, N).

```



# APPENDIX A.3 (Continued)

```

621 length(Bizarre, _) :- error(length(Bizarre, _)).
622
623 % bind every variable to a distinct 'V'(N)
624 numbervars('V'(N), N, NextN) :- !, sum(N, 1, NextN).
625 numbervars('V'(_), N, N) :- !.
626 numbervars(X, N, N) :- integer(X), !.
627 numbervars(X, N, NextN) :- numbervars(X, 1, N, NextN).
628
629 numbervars(X, K, N, NextN) :-
630     arg(K, X, A), !, numbervars(A, N, MidN),
631     sum(K, 1, K1), numbervars(X, K1, MidN, NextN).
632 numbervars(_, _, N, N).
633
634 % .....
635 % predefined "fix" functors and op
636 % .....
637 % (ordered according to probable frequency)
638 'FF'(';', [xfy], 1000).
639 'FF'(':', [xfx, fx], 1200).
640 'FF'(':', [xfy], 1100).
641 'FF'(not, [fy], 900).
642 'FF'(=, [xfx], 700).
643 'FF'(is, [xfx], 700).
644 'FF'(-->, [xfx], 1200).
645 'FF'(+, [yfx, fx], 500). 'FF'(-, [yfx, fx], 500).
646 'FF'(*, [yfx], 400). 'FF'(/, [yfx], 400).
647 'FF'(mod, [xfx], 300).
648 'FF'(<, [xfx], 700). 'FF'(<=, [xfx], 700).
649 'FF'(>, [xfx], 700). 'FF'(>=, [xfx], 700).
650 'FF'(:=, [xfx], 700). 'FF'(==, [xfx], 700).
651 'FF'(@<, [xfx], 700). 'FF'(@<=, [xfx], 700).
652 'FF'(@>, [xfx], 700). 'FF'(@>=, [xfx], 700).
653 'FF'(:=, [xfx], 700).
654 'FF'(==, [xfx], 700). 'FF'(==, [xfx], 700).
655
656 % this implementation of op takes care of redefinitions
657 % and of mixed functors
658 op(Prior, Type, Name) :-
659     atom(Name), pname(Name, String), noq(String),
660     % noq - see WRITE
661     integer(Prior), less(0, Prior), less(Prior, 1201),
662     set_kind(Type, Kind), !,
663     do_op(Prior, Type, Name, Kind).
664 % if not all parameters are OK -
665 op(P, T, N) :- error(op( P, T, N )).
666
667 % set Kind to bin or un
668 set_kind(Type, bin) :- binary(Type, _), !.
669 set_kind(Type, un) :- unary(Type, _, _), !.
670
671 % test for binary and instantiate Assoc
672 binary(xfy, a(r)). % right associative

```



```

673 binary(yfx, a(l)). % left associative
674 binary(xfx, na(_)). % non-associative
675 % test for unary, instantiate Kind and Assoc
676 unary(fy, pre, a(r)). % right associative
677 unary(fx, pre, na(r)). % right non-associative
678 unary(yf, post, a(l)). % left associative
679 unary(xf, post, na(l)). % left non-associative
680
681 do_op(P, T, N, Kind) :-
682     'FF'(N, Oldtypes, Oldprior), !,
683     addff(Oldtypes, Oldprior, P, T, N, Kind).
684 do_op(P, T, N, _) :- assertz('FF'(N, [T], P)).
685
686 % add or redefine a functor
687 % for mixed functors, keep the binary type before the unary
688
689 % the same priority: redefine or make mixed
690 addff([Oldtype], P, P, T, N, Kind) :-
691     !, set kind(Oldtype, Oldkind),
692     addff1(Oldkind, Kind, Oldtype, T, N, P).
693 addff([Oldtype1, Oldtype2], P, P, T, N, Kind) :-
694     !, addff2(Kind, Oldtype1, Oldtype2, T, P, N).
695 % otherwise the priorities were different: redefine
696 addff(_, _, P, T, N, _) :- redef(N, [T], P).
697
698 % make a mixed functor or change type
699 addff1(bin, un, Oldtype, T, N, P) :- mk_mixed(N, [Oldtype, T], P).
700 addff1(un, bin, Oldtype, T, N, P) :- mk_mixed(N, [T, Oldtype], P).
701 addff1(Kind, Kind, _, T, N, P) :- redef(N, [T], P).
702
703 % adjust a mixed functor by changing one of its types
704 addff2(bin, _, Oldtype2, T, P, N) :- mk_mixed(N, [T, Oldtype2], P).
705 addff2(un, Oldtype1, _, T, P, N) :- mk_mixed(N, [Oldtype1, T], P).
706
707 mk_mixed(N, Types, P) :-
708     retract('FF'(N, _, _)), !, assertz('FF'(N, Types, P)).
709
710 % redefine and issue a warning
711 redef(N, T, P) :-
712     nl, display('functor '), display(N),
713     display(' redefined'), nl,
714     retract('FF'(N, _, _)), !, asserta('FF'(N, T, P)).
715
716 % remove a declaration
717 delop(Name) :- atom(Name), retract('FF'(Name, _, _)), !.
718 delop(Name) :- error(delop(Name)).
719
720
721 % .....
722 % evaluate an arithmetic expression
723 % .....
724 is(N, N) :- integer(N), !.

```



```

725 is(Val, +(A, B)) :-
726     !, is(Av, A), is(Bv, B), sum(Av, Bv, Val).
727 is(Val, -(A, B)) :-
728     !, is(Av, A), is(Bv, B), sum(Bv, Val, Av).
729 is(Val, *(A, B)) :-
730     !, is(Av, A), is(Bv, B), prod(Av, Bv, 0, Val).
731 is(Val, /(A, B)) :-
732     !, is(Av, A), is(Bv, B), prod(Bv, Val, _, Av).
733 is(Val, mod(A, B)) :-
734     !, is(Av, A), is(Bv, B), prod(Bv, _, Val, Av).
735 is(Val, +(A)) :- !, is(Val, A).
736 is(Val, -(A)) :- !, is(Av, A), sum(Val, Av, 0).
737 is(N, [N]) :- integer(N).
738 % otherwise fail
739
740 % ----- EVALUATE AN ARITHMETIC RELATION -----
741 ===(X, Y) :- is(XV, X), is(YV, Y).
742 <(X, Y) :- is(XV, X), is(YV, Y), less(XV, YV).
743 <=(X, Y) :- is(XV, X), is(YV, Y), not(less(YV, XV)).
744 >(X, Y) :- is(XV, X), is(YV, Y), less(YV, XV).
745 >=(X, Y) :- is(XV, X), is(YV, Y), not(less(XV, YV)).
746 ===(X, Y) :- not(===(X, Y)).
747
748 % .....
749 %           perfect equality of terms
750 % .....
751 ==(T1, T2) :- var(T1), var(T2), !, eqvar(T1, T2).
752 ==(T1, T2) :- check(==(T1, T2)).
753
754 ==(T1, T2) :- not(==(T1, T2)).
755
756 ==?(T1, T2) :-
757     integer(T1), integer(T2), !, =(T1, T2).
758 ==?(T1, T2) :-
759     nonvarint(T1), nonvarint(T2),
760     functor(T1, Fun, Arity), functor(T2, Fun, Arity),
761     equalargs(T1, T2, 1).
762
763 equalargs(T1, T2, Argnumber) :-
764     arg(Argnumber, T1, Arg1), arg(Argnumber, T2, Arg2),
765     % arg fails given too large a number
766     !, ==(Arg1, Arg2), sum(Argnumber, 1, Nextnumber),
767     equalargs(T1, T2, Nextnumber).
768 equalargs(_, _, _).
769
770 % .....
771 %   assert, asserta, assertz, retract, clause
772 % .....
773 % - - - add a clause (using built-in assert(_, _))
774 assert(Cl) :- asserta(Cl).
775 asserta(Cl) :-
776     nonvarint(Cl), convert(Cl, Head, Body), !,

```

```

777     assert(Head, Body, 0).
778 asserta(Cl) :- error(asserta(Cl)).
779
780 assertz(Cl) :-
781     nonvarint(Cl), convert(Cl, Head, Body), !,
782     assert(Head, Body, 32767).    % ie 2 to 15th minus 1
783 assertz(Cl) :- error(assertz(Cl)).
784
785 % convert the external form of a Body into a dotted list
786 convert(-(Head, B), Head, Body) :- conv_body(B, Body).
787 convert(Unit_cl, Unit_cl, []).
788
789 % this procedure works both ways
790 conv_body(B, [call(B)]) :- var(B), !.
791 conv_body(true, []).
792 conv_body(B, Body) :- conv_b(B, Body).
793
794 conv_b(B, [Body]) :- var(B), !, conv_call(B, Body).
795 conv_b(',(C, B), [Call | Body]) :-
796     !, conv_call(C, Call), conv_b(B, Body).
797 conv_b(Call, [Call]).    % not a variable
798
799 % interpreter can process variable calls only within c a l l
800 conv_call(C, call(C)) :- var(C), !.
801 conv_call(C, C).
802
803 % - - - remove a clause (this procedure is backtrackable)
804 retract(Cl) :-
805     nonvarint(Cl), convert(Cl, Head, Body), !,
806     functor(Head, Fun, Arity), remcls(Fun, Arity, 1, Head, Body).
807 retract(Cl) :- error(retract(Cl)).
808
809 % ultimate failure if N too big (retract/3 fails)
810 remcls(Fun, Arity, N, Head, Body) :-
811     clause(Fun, Arity, N, N_head, N_body),
812     remcls(Fun, Arity, N, N_head, Head, N_body, Body).
813
814 remcls(Fun, Arity, N, Head, Head, Body, Body) :-
815     retract(Fun, Arity, N).
816 % user's backtracking resumes r e t r a c t here
817 % (after removing the Nth clause the next becomes Nth)
818 remcls(Fun, Arity, N, N_head, Head, N_body, Body) :-
819     check(=(N_head, Head)), check(=(N_body, Body)),
820     !, remcls(Fun, Arity, N, Head, Body).
821 remcls(Fun, Arity, N, _, Head, _, Body) :-
822     sum(N, 1, N1), remcls(Fun, Arity, N1, Head, Body).
823
824 % - - - generate nondeterministically all clauses whose head
825 % and body match the parameters of c l a u s e
826 clause(Head, Body) :-
827     nonvarint(Head), !, functor(Head, Fun, Arity),
828     gensls(Fun, Arity, 1, Head, Body).

```



# APPENDIX A.3 (Continued)

```

829 clause(Head, Body) :- error(clause(Head, Body)).
830
831 % generate; ultimate failure if N too big (clause/5 fails)
832 genscls(Fun, Arity, N, Head, Body) :-
833     clause(Fun, Arity, N, N_head, N_body),
834     genscls(Fun, Arity, N, N_head, Head, N_body, Body).
835
836 % fail if N_head does not match Head,
837 % or if N_body converted does not match Body
838 genscls(_, _, N_head, N_head, N_body, Body) :-
839     conv_body(Body, N_body).
840 % user's backtracking resumes clause here
841 genscls(Fun, Arity, N, _, Head, _, Body) :-
842     sum(N, 1, N1), genscls(Fun, Arity, N1, Head, Body).
843
844 % .....
845 % listing
846 % .....
847 % list procedures determined by the parameter ( listing( ) )
848 % or all user's procedures ( listing )
849 listing :-
850     proc(Head), listproc(Head), nl, fail.
851 listing. % catch the final fail from p r o c
852
853 listing(Fun) :- atom(Fun), !, listbyname(Fun).
854 listing(/(Fun, Arity)) :-
855     atom(Fun), integer(Arity), <=(0, Arity), !,
856     functor(Head, Fun, Arity), listproc(Head).
857 listing(L) :-
858     isclosedlist(L), listseveral(L), !.
859 listing(X) :- error(listing(X)).
860 % isclosedlist - cf grammar rule preprocessor
861
862 listseveral([]).
863 listseveral([Item | Items]) :-
864     listing(Item), listseveral(Items).
865
866 % all procedures with this name
867 listbyname(Fun) :-
868     proc(Head), functor(Head, Fun, _),
869     listproc(Head), nl, fail.
870 listbyname(_). % succeed
871
872 % one procedure
873 listproc(Head) :-
874     clause(Head, Body),
875     writeclause(Head, Body), wch(., nl, fail.
876 listproc(_). % succeed
877
878 writeclause(Head, Body) :-
879     not(var(Body)), =(Body, true), !, writeq(Head).
880 writeclause(Head, Body) :- writeq(:-(Head, Body)).

```

```

881
882 % .....
883 %      write
884 % .....
885 write(Term) :- side_effects(outterm(Term, noq)).
886
887 % writeq encloses in quotes all identifiers except words,
888 % symbols and solochars (not coinciding with "fix" functors)
889 writeq(Term) :- side_effects(outterm(Term, q)).
890
891 writetext([Ch | Chs]) :- !, wch(Ch), writetext(Chs).
892 writetext([]).
893
894 outterm(T, Q) :- numbervars(T, 1, _), outt(T, fd(_, _), Q).
895
896 % the real job is done here
897 outt('V'(N), _, _) :- integer(N), !, wch('X'), display(N).
898 % C A U T I O N : outt is unable to write 'V'(Integer)
899 outt(Term, _, _) :- integer(Term), display(Term), !.
900 % the second parameter specifies a context for "fix" functors:
901 % the nearest external functor and Term's position
902 % (to the left or to the right of the external functor)
903 outt(Term, Context, Q) :-
904     =..(Term, [Name | Args]),
905     outfun(Name, Args, Context, Q).
906
907 % - - - output a functor-term
908 % - as a "fix" term
909 outfun(Name, Args, Context, Q) :-
910     isfix(Name, Args, This_ff, Kind), !,
911     outff(Kind, This_ff, [Name | Args], Context, Q).
912 % - as a list
913 outfun(_, [Larg, Rarg], _, Q) :-
914     !, outlist([Larg | Rarg], Q).
915 % - as a normal functor-term
916 outfun(Name, Args, _, Q) :-
917     outname(Name, Q), outargs(Args, Q).
918
919 % isfix constructs a pair ff(Prior, Associativity) , and
920 % 'in' or 'pre' or 'post' (fails if not a "fix" functor)
921 isfix(Name, [_], ff(Prior, Assoc), in) :-
922     'FF'(Name, Types, Prior), mk_bin(Types, Assoc).
923 isfix(Name, [_], ff(Prior, Assoc), Kind) :-
924     'FF'(Name, Types, Prior), mk_un(Types, Kind, Assoc).
925
926 % Bintype (if any) is before Untype (if any)
927 mk_bin([Bintype | _], Assoc) :- binary(Bintype, Assoc).
928 mk_un([Untype], Kind, Assoc) :- unary(Untype, Kind, Assoc).
929 mk_un([_, Untype], Kind, Assoc) :- unary(Untype, Kind, Assoc).
930 % tests - see o p
931
932 % - - - output a "fix" term (this outff has 5 parameters)

```



```

933 outff(Kind, This_ff, NameArgs, Context, Q) :-
934     agree(This_ff, Context), !,
935     outff(Kind, This_ff, NameArgs, Q).
936 outff(Kind, This_ff, NameArgs, _ Q) :-
937     wch('('), outff(Kind, This_ff, NameArgs, Q), wch(')').
938
939 % agree helps avoid (some) unnecessary brackets around the term
940 agree(_, fd(Ext_ff, _)) :- var(Ext_ff).
941 agree(ff(Prior1, _), fd(ff(Prior2, _), _)) :-
942     stronger(Prior1, Prior2). % cf the parser
943 agree(ff(Prior, a(Dir)), fd(ff(Prior, a(Dir)), Dir)).
944
945 % output the functor and the arguments (this outff has 4 parameters)
946 outff(in, This_ff, [Name, Larg, Rarg], Q) :-
947     out(Larg, fd(This_ff, l), Q),
948     outfn(Name, ' '), out(Rarg, fd(This_ff, r), Q).
949 outff(pre, This_ff, [Name, Arg], Q) :-
950     outfn(Name, ' '), out(Arg, fd(This_ff, r), Q).
951 outff(post, This_ff, [Name, Arg], Q) :-
952     out(Arg, fd(This_ff, l), Q), outfn(Name, ' ').
953
954 % output functor's name enclosed in Encl
955 outfn(Name, Encl) :- wch(Encl), display(Name), wch(Encl).
956
957 % - - - print a name (in quotes, if necessary)
958 outname(Name, noq) :- !, display(Name).
959 outname(Name, q) :-
960     'FF'(Name, _ , _), !, outfn(Name, "").
961 outname(Name, q) :-
962     pname(Name, Namestring),
963     check(noq(Namestring)), !, display(Name).
964 outname(Name, q) :- outfn(Name, "").
965
966 noq([Ch | String]) :- wordstart(Ch), isword(String).
967 noq([Ch]) :- solochar(Ch).
968 noq(['[', ']').
969 noq([Ch | String]) :- symch(Ch), issym(String).
970
971 isword([]).
972 isword([Ch | String]) :- alphanum(Ch), isword(String).
973 issym([]).
974 issym([Ch | String]) :- symch(Ch), issym(String).
975
976 % - - - output a list of arguments (cf outfun)
977 outargs([], _) :- !.
978 outargs(Args, Q) :-
979     fake(Context), wch('('), outargs(Args, Context, Q), wch(')').
980
981 outargs([Last], Context, Q) :- !, out(Last, Context, Q).
982 outargs([Arg | Args], Context, Q) :-
983     out(Arg, Context, Q), display(', '), outargs(Args, Context, Q).
984

```

### APPENDIX A.3 (Continued)

```

985 % commas are used to delimit list items, so we must bracket commas
986 %   within items (it's a trick: we depend on ',' having
987 %   the priority 1000 and being associative)
988 fake(fd(ff(1000, na(_), _)).
989
990 % - - - output a list in square brackets (cf outfun - the main
991 %   functor is the dot, and the list cannot be empty)
992 outlist([First | Tail], Q) :-
993     fake(Context), wch('['), outt(First, Context, Q),
994     outlist(Tail, Context, Q), wch(']').
995
996 outlist([], _, _) :- !.
997 outlist([Item | Items], Context, Q) :-
998     !, display(' '), outt(Item, Context, Q),
999     outlist(Items, Context, Q).
1000 % the bar and the closing item (still bracketed if it contains commas)
1001 outlist(Closing, Context, Q) :-
1002     display(' | '), outt(Closing, Context, Q).
1003
1004 % *****
1005 % *****
1006 %   t r a n s l a t o r
1007 % *****
1008 % *****
1009 % read a program upto end. and translate it into "kernel" form
1010 translate(Infile, Outfile) :-
1011     see(Infile), tell(Outfile),
1012     nl, repeat,
1013     read(Clause), put(Clause), nl, =(Clause, end), !,
1014     seen, told, see(user), tell(user).
1015
1016 % - - - produce and output the translation of one clause
1017 put(-(Head, Body)) :-
1018     !, puthead(Head, Sym_tab), putbody(Body, Sym_tab).
1019 put(-->(Left, Right)) :-
1020     !, tag(transl_rule(Left, Right, -(Head, Body))),
1021     puthead(Head, Sym_tab), putbody(Body, Sym_tab).
1022 put(-(Goal)) :-
1023     !, putbody(Goal, Sym_tab), wch('#'), nl,
1024     once(Goal). % a failure here wouldn't matter (cf translate)
1025 put(end) :- !.
1026 put('e r r') :- !.
1027 put(Unitclause) :- puthead(Unitclause, Sym_tab), putbody(true, _).
1028
1029 % - - - put a head call (it must be a functor-term)
1030 puthead(Head, Sym_tab) :-
1031     nonvarint(Head), !, putterm(Head, Sym_tab).
1032 puthead(Head, _) :- transl_err(Head).
1033
1034 % - - - put a list of calls and [] at the end
1035 putbody(Body, Sym_tab) :-
1036     punct(:), conv_body(Body, B), !, putbody_c(B, Sym_tab).

```



### APPENDIX A.3 (Continued)

---

```

1037          % see assert etc for c o n v _ b o d y
1038
1039 putbody_c([], _) :- !, display([]).
1040 putbody_c([Term | Terms], Sym_tab) :-
1041     not(integer(Term)), !, putterm(Term, Sym_tab),
1042     punct(.), putbody_c(Terms, Sym_tab).
1043 putbody_c([Term | _], _) :- transl_err(Term).
1044
1045 punct(Ch) :- wch(' '), wch(Ch), nl, display(' ').
1046
1047 % - - - put a term (with infix dots, and canonical otherwise)
1048 putterm(Term, Sym_tab) :-
1049     var(Term), !, lookup(Term, Sym_tab, -1, N),
1050     wch(:), display(N).
1051 putterm(Term, _) :- integer(Term), !, display(Term).
1052 putterm([Head | Tail], Sym_tab) :-
1053     !, putterm_inlist(Head, Sym_tab),
1054     display(' '), putterm(Tail, Sym_tab).
1055 putterm(Term, Sym_tab) :-
1056     =..(Term, [Name | Args]), outfn(Name, ''), % cf WRITE
1057     putargs(Args, Sym_tab).
1058
1059 % Sym_tab is an open list of pairs vn(Variable, Number)
1060 % (this formulation helps avoid too many additions)
1061 lookup(V, S_t_end, PreviousN, N) :-
1062     var(S_t_end), !, sum(PreviousN, 1, N),
1063     =(S_t_end, [vn(V, N) | New_s_t_end]).
1064 lookup(V, [vn(CurrV, CurrN) | _, _, CurrN] :-
1065     eqvar(V, CurrV), !.
1066 lookup(V, [vn(_, CurrN) | S_t_tail], _, N) :-
1067     lookup(V, S_t_tail, CurrN, N).
1068
1069 % arguments - nothing, or a list of terms in parentheses
1070 putargs([], _) :- !.
1071 putargs(Args, Sym_tab) :-
1072     wch('('), putarglist(Args, Sym_tab), wch(')').
1073
1074 putarglist([Arg], Sym_tab) :- !, putterm(Arg, Sym_tab).
1075 putarglist([Arg | Args], Sym_tab) :-
1076     putterm(Arg, Sym_tab), display(' '),
1077     putarglist(Args, Sym_tab).
1078
1079 % - - - a list within a list must be enclosed in parentheses
1080 putterm_inlist(Term, Sym_tab) :-
1081     nonvarint(Term), =(Term, [_ | _]), !,
1082     wch('('), putterm(Term, Sym_tab), wch(')').
1083 putterm_inlist(Term, Sym_tab) :- putterm(Term, Sym_tab).
1084
1085 % - - - error handling (only one error is discovered by translate)
1086 transl_err(X) :-
1087     nl, display('+++ Bad head or call: '), display(X), nl, fail.
1088
1089 :- see(user), ear.

```

## APPENDIX A.4

### Three Useful Programs

---

#### A simple editor

---

```
% A simple interactive clause editor.
% Watch for name conflicts with its procedures !
% Note that this version has no safeguards against Prolog's crash
% (eg. due to stack overflow).
% Call edit( Name/arity ) to edit the procedure of this name and arity.
% Each invocation of edit is associated with a cursor, which is the number
% of a clause. Initially the cursor is at clause 0, i.e. before the first
% clause in this procedure. The cursor's value and its associated clause
% is usually displayed between commands.
% Commands are listed below. Terminate the line immediately after typing
% last character. Don't use blanks where not shown and only one where shown.
%
% Commands :
% _____
%
% e Name/Arity - invoke a nested instance to edit another procedure.
%               The current cursor stays in place unless you happen
%               to modify this procedure within a nested instance.
%
% x           - exit from the current editor instance.
%
% +           - move the cursor to the next clause, no action if none.
%
% <cr>        - an empty line is an alternative form of +.
%
% -           - move the cursor to the previous clause, no action if at 0.
%
% t           - top : move the cursor to 0.
%
% b           - bottom : move the cursor to the bottom clause
%               (0 for empty procedures).
%
% l           - list the whole procedure.
%
% d           - delete the current clause and move the cursor to
%               the next (or to the new bottom if bottom is deleted).
%
% i           - insert after the current clause. In the following
%               lines write clauses as you would after consult(user)
%               (terminate the sequence with end.). The cursor is
%               positioned at the last inserted clause.
%
% f Filename  - like i, but read the clauses from a file.
%               Take care ! filename correctness is not checked.
%
% p           - invoke a nested instance of Prolog. If there is
%               no memory overflow, invoking stop will return
%               control to the editor.
%
% _____
```

```
edit( Name/Arity ) :- not ( atom( Name ), integer( Arity ) ), !,
    write( 'Bad parameters : ' ),
    write( edit( Name/Arity ) ), nl, fail.
edit( Name/Arity ) :- predefined( Name, Arity ), !,
    write( 'Can't edit system routine : ' ),
    write( Name/Arity ), nl, fail.
edit( NameArity ) :- tag( ed( NameArity, 0 ) ).
```



---

```

ed( NameArity, Cursor ) :- show( NameArity, Cursor ), !,
                           docmd( NameArity, Cursor, NewCursor ),
                           ed( NameArity, NewCursor ).
ed( NameArity, Cursor ) :- display( 'Cursor out of range : ' ),
                           display( Cursor ), nl, ed( NameArity, 0 ).
docmd( NameArity, Cursor, NewCursor ) :-
    repeat, % repeat over incorrect commands
    getline( Line ), cmd( Line, NameArity, Cursor, NewCursor ),
    !.

getline( [] ) :- rch, lastch( C ), iscoln( C ), !.
getline( [ C | L ] ) :- lastch( C ), getline( L ).

% cmd fails for incorrect commands.
cmd( [], NmAr, Cur, NCur ) :- next_cursor( NmAr, Cur, NCur ).
cmd( ['+', NmAr, Cur, NCur ) :- next_cursor( NmAr, Cur, NCur ).
cmd( ['-'], NmAr, Cur, NCur ) :- prev_cursor( Cur, NCur ).
cmd( [t], NmAr, _, 0 ).
cmd( [b], NmAr, Cur, NCur ) :- bottom_cursor( NmAr, Cur, NCur ).
cmd( [l], NmAr, Cur, Cur ) :- listing( NmAr ).
cmd( [d], NmAr, Cur, NCur ) :- delete( NmAr, Cur, NCur ).
cmd( [i], NmAr, Cur, NCur ) :- insert( NmAr, Cur, NCur ).
cmd( [f, ' ' | NameString], NmAr, Cur, NCur ) :-
    file_insert( NameString, NmAr, Cur, NCur ).
cmd( [e, ' ' | Args], NmAr, Cur, Cur ) :-
    append( NameString, [' ' | ArityString], Args ),
    call_edit( NameString, ArityString ).
cmd( [x], _, _, _ ) :- tagexit( ed( _, _ ) ).
cmd( [p], NmAr, Cur, Cur ) :- invoke_Prolog.
cmd( String, _, _, _ ) :- display( '----incorrect command : ' ),
    writetext( String ), nl, fail.

% check is provided with the standard library ( check(C) :- not not C )
next_cursor( Name/Arity, Cursor, Next ) :-
    Next is Cursor + 1, check( clause( Name, Arity, Next, _, _ ) ), !.
next_cursor( _, Cursor, Cursor ). % cursor at last clause

prev_cursor( 0, 0 ).
prev_cursor( Cursor, Prev ) :- Cursor > 0, Prev is Cursor - 1.

bottom_cursor( Name/Arity, Cursor, Bottom ) :-
    Next is Cursor + 1, check( clause( Name, Arity, Next, _, _ ) ),
    !, bottom_cursor( Name/Arity, Next, Bottom ).
bottom_cursor( _, Cursor, Cursor ).

delete( _, 0, 0 ) :- !, display( 'Can't delete clause 0' ), nl.
delete( Name/Arity, Cursor, NewCursor ) :-
    retract( Name, Arity, Cursor ),
    cursor_in_range( Name, Arity, Cursor, NewCursor ).

cursor_in_range( Nm, Ar, Cur, Cur ) :-
    check( clause( Nm, Ar, Cur, _, _ ) ), !.

```

## APPENDIX A.4 (Continued)

cursor\_in\_range( \_, \_, Cur, Prev ) :- Prev is Cur - 1.

% convert is defined in the standard library

insert( NameAriy, Cursor, NewCursor ) :-

repeat, % get end. or a clause of Name/Ariy, skip others

read( Clause ), convert( Clause, Head, Body ),

accept( Head, NameAriy, Clause ),

!,

end\_or\_proceed( Head, Body, NameAriy, Cursor, NewCursor ).

end\_or\_proceed( end, [], \_, Cursor, Cursor ) :- !.

end\_or\_proceed( Head, Body, NameAriy, Cursor, NewCursor ) :-

Next is Cursor + 1, assert( Head, Body, Cursor ),

insert( NameAriy, Next, NewCursor ).

accept( \_, \_, end ).

accept( Head, Name/Ariy, \_ ) :- functor( Head, Name, Ariy ).

accept( \_, \_, Clause ) :-

display( '---clause not in edited procedure - ignored' ),

nl, write( Clause ), fail.

file\_insert( FNameString, NameAriy, Cursor, NewCursor ) :-

pname( FileName, FNameString ),

see( FileName ), insert( NameAriy, Cursor, NewCursor ),

seen, see( user ).

call\_edit( NameString, AriyString ) :-

pname( Name, NameString ), pname( Ariy, AriyString ),

edit( Name/Ariy ).

invoke\_Prolog :- tag( loop ), % this works only for the Toy-Prolog monitor

invoke\_Prolog. % ( loop terminated by tagfail )

% conv\_body is defined in the standard library (asserta etc.),

% so is writeclause.

show( NameAriy, 0 ) :- !, write( '[0] ( ' ), write( NameAriy ),

write( ' )' ), nl.

show( Name/Ariy, Cursor ) :-

side\_effects( ( clause( Name, Ariy, Cursor, Head, Body ),

conv\_body( NiceBody, Body ),

display( 'T' ), display( Cursor ),

display( ' ' ),

writeclause( Head, NiceBody ),

display( '.' ), nl ).

append( [], L, L ).

append( [ E | L ], L2, [ E | LL2 ] ) :- append( L, L2, LL2 ).



**A primitive tracing tool**

```

% A primitive tracing package.
% Watch for name conflicts with its procedures !
% Use spy( Pattern ) to trace calls matching Pattern,
%   nospy( Pattern ) to stop tracing.
% To trace, execute trace( Goal ) instead of Goal.
% Successful calls are displayed with a plus, failing calls with a minus.
% Note: tagcut, tagexit, tagfail and ancestor will not be executed properly.
%   trace is slow : if you wish to have the insides of a correct and
%   costly procedure executed at normal speed, add
%   a predefined(...) assertion for its call.

spy( All ) :- var( All ), !, assert( spied( All ) ).
spy( Pattern ) :- spied( Pattern ), !.           % spied already
spy( Pattern ) :- assert( spied( Pattern ) ).

nospy( Pattern ) :- retract( spied( Pattern ) ), fail.
nospy( _ ).

trace( Goal ) :- tag( runbody( Goal ) ).

runbody( A , B ) :- !, runbody( A ), runbody( B ).
runbody( A ; B ) :- !, ( runbody( A ) ; runbody( B ) ).
runbody( call( Call ) ) :-
    var( Call ), !, showfailure( call( Call ) ), fail.
runbody( call( Call ) ) :- !, runbody( Call ).
runbody( tag( Call ) ) :- !, runbody( call( Call ) ).
runbody( Call ) :- predefined( Call ), !, runsystem( Call ).
runbody( Call ) :- tag( runuser( Call ) ).

runsystem( ! ) :- runcut.
runsystem( Forbidden ) :-
    isforbidden( Forbidden ), !, nl,
    display( 'FORBIDDEN CALL ' ), write( Forbidden ),
    display( ' FAILS !' ), nl, fail.
runsystem( Call ) :- not spied( Call ), !, Call.
runsystem( Call ) :- Call, !, showsuccess( Call ).
runsystem( Call ) :- showfailure( Call ), fail.

runuser( Call ) :- not spied( Call ), !,
    clause( Call, Body ), runbody( Body ).
runuser( Call ) :- clause( Call, Body ), showsuccess( Call ),
    runbody( Body ).
runuser( Call ) :- showfailure( Call ), fail.

runcut :- spied( ! ), simulatecut, showsuccess( ! ).
runcut :- simulatecut.

simulatecut :- tagcut( runuser( _ ) ).
simulatecut :- tagcut( runbody( _ ) ).    % cut in initial goal

```

**showsucces**( Call ) :- display( ' + ' ), write( Call ), nl.

**showfailure**( Call ) :- display( ' - ' ), write( Call ), nl.

**isforbidden**( tagexit( \_ ) ).

**isforbidden**( tagfail( \_ ) ).

**isforbidden**( tagcut( \_ ) ).

**isforbidden**( ancestor( \_ ) ).

**predefined**( Call ) :-

**check**( ( functor( Call, F, N ), predefined( F, N ) ) ).



## A program structure analyser with analyser analysed

% Given a procedure name and arity, print its call tree.  
 % The main data structure is a queue of procedures whose tail contains  
 % calls which were not yet seen. Each element of the queue contains a list  
 % of calls (references to main queue elements) and a variable to hold its  
 % ordinal number in the listed tree.  
 % Queues are searched linearly : the algorithm is costly for large trees.  
 % CAUTION : don't attempt to list a trace of this program - cyclic structures  
 % are formed as a rule.

```
calltree( Name/Arity ) :- add( proc( Name, Arity, Ord, Calls ), Queue ),
                          fill( Queue, Queue ),
                          print_calls([proc(Name,Arity,Ord,Calls)],3,1,_).
```

```
% add finds (inserts) an element in ( to ) an open list
add( El, [El | Tail] ) :- !.
add( El, [ _ | Tail ] ) :- add( El, Tail ).
```

```
% fill walks the queue and expands procedures, inserting their calls into
% the queue if not yet seen. Queue beginning is passed along to allow search.
fill( [], _ ) :- !. % evidently reached the terminating variable
fill( [proc(Name,Arity,_,[])|QTail], Q ) :- predefined( Name / Arity ), !,
                                           fill( QTail, Q ).
fill( [proc(Name,Arity,_,undefined)|QTail], Q ) :-
    not clause( Name, Arity, 1, _, _ ), !, fill( QTail, Q ).
fill( [proc(Name,Arity,_,Calls)|QTail], Q ) :-
    add_calls( Name, Arity, 1, Calls, Q ), fill( QTail, Q ).
```

```
% system procedures and procedures defined in the monitor should not be shown
predefined( Name / Arity ) :- predefined( Name, Arity ).
% only the more commonly used procedures (but the list is easily extended)
predefined( 'not' / 1 ). predefined( 'nl' / 0 ).
predefined( read / 1 ). predefined( '=' / 2 ).
predefined( op / 3 ). predefined( 'is' / 2 ).
predefined( assert / 1 ). predefined( assertz / 1 ).
predefined( retract / 1 ). predefined( clause / 2 ).
predefined( write / 1 ). predefined( writeq / 1 ).
```

```
% add_calls processes the clauses of a procedure, adding calls to its list
% of calls and to the queue (only finding in the queue if already there)
add_calls( Name, Arity, N, Calls, Q ) :-
    clause( Name, Arity, N, _, Body ), !,
    body_calls( Body, Calls, Q ), N1 is N + 1,
    add_calls( Name, Arity, N1, Calls, Q ).
add_calls( _, _, _, [], _ ) :- !. % close the list if empty
add_calls( _, _, _, _, _ ) :- !. % ( only unit clauses )
                                % non-empty list left open
body_calls( [], _, _ ) :- !.
```

## APPENDIX A.4 (Continued)

body\_calls( [Call | BodyTail], Calls, Q ) :-

```
    functor( Call, Name, Arity ),
    add( proc(Name,Arity,Ord,Callees), Calls ),
    add( proc(Name,Arity,Ord,Callees), Q ),
    add_insidess( Call, Calls, Q ),
    body_calls( BodyTail, Calls, Q ).
```

% add\_insidess unpacks metalogical calls: if their arguments are not variable  
% or integer, they are added to the queues.

```
add_insidess( Call, Q1, Q2 ) :- meta_call_1( Call, Arg ), !,
    add_inside( Arg, Q1, Q2 ).
```

```
add_insidess( Call, Q1, Q2 ) :- meta_call_2( Call, Arg1, Arg2 ), !,
    add_inside( Arg1, Q1, Q2 ),
    add_inside( Arg2, Q1, Q2 ).
```

```
add_insidess( _, _, _ ).
```

```
add_inside( V, _, _ ) :- ( var( V ) ; integer( V ) ), !.
```

```
add_inside( Call, Q1, Q2 ) :- functor( Call, Name, Arity ),
    add( proc(Name,Arity,Ord,Callees), Q1 ),
    add( proc(Name,Arity,Ord,Callees), Q2 ),
    add_insidess( Call, Q1, Q2 ).
```

```
meta_call_1( call( Call ), Call ).
```

```
meta_call_1( tag( Call ), Call ).
```

```
meta_call_1( not Call, Call ).
```

```
meta_call_1( check( Call ), Call ).
```

```
meta_call_1( side_effects( Call ), Call ).
```

```
meta_call_1( once( Call ), Call ).
```

```
meta_call_2( ( A , B ), A, B ).
```

```
meta_call_2( ( A ; B ), A, B ).
```

% Print calls, starting at given tab setting and ordinal, returning next ordinal  
% number. Third clause fails if ordinal numbers don't match, i.e. proc  
% was already printed in another line.

```
print_calls( [], _, Ord, Ord ) :- !. % this matches the terminating var  
    % of a call list.
```

```
print_calls( [proc(Name,Arity,Ord,undefined)|Calls], Tab, Ord, NOrd ) :-
    !, start_undefined( Ord, Tab ),
    writeq( Name/Arity ), display( ' **undefined***' ), nl,
    TOrd is Ord + 1, print_calls( Calls, Tab, TOrd, NOrd ).
```

```
print_calls( [proc(Name,Arity,Ord,Callees)|Calls], Tab, Ord, NOrd ) :-
    !, start_line( Ord, Tab ), writeq( Name/Arity ), nl,
    InnerTab is Tab + 3, InnerOrd is Ord + 1,
    print_calls( Callees, InnerTab, InnerOrd, TOrd ),
    print_calls( Calls, Tab, TOrd, NOrd ).
```

```
print_calls( [proc(Name,Arity,AnotherOrd,_)|Calls], Tab, Ord, NOrd ) :-
    start_unnumbered_line( Tab ), writeq( Name/Arity ),
    repetition( Name, Arity, AnotherOrd ), nl,
    print_calls( Calls, Tab, Ord, NOrd ).
```



```

repetition( Name, Arity, _ ) :- predefined( Name, Arity ), !.
repetition( _, _, Ord ) :- display( ' (see ' ), display( Ord ),
                             display( ')' ).

```

```

% Ord numbers are printed in 4 columns, right justified
start_line( Ord, Tab ) :- number_line( Ord ), !, tab( Tab, '' ).
number_line( N ) :- N < 10, display( ' ' ), display( N ).
number_line( N ) :- N < 100, display( ' ' ), display( N ).
number_line( N ) :- N < 1000, display( ' ' ), display( N ).
number_line( N ) :- display( N ).

```

```

start_unnumbered_line( Tab ) :- display( ' ' ), tab( Tab, '' ).

```

```

start_undefined( Ord, Tab ) :- number_line( Ord ), tab( Tab, '' ).

```

```

tab( 0, _ ) :- !.
tab( N, Ch ) :- wch( Ch ), N1 is N - 1, tab( N1, Ch ).
?- % -----

```

```

% a sample call and results

```

```

:- calltree( calltree / 1 ).
1  calltree / 1
2  add / 2
3  ! / 0
   add / 2 (see 2)
4  fill / 2
   ! / 0
5  predefined / 1
6  predefined / 2
   fill / 2 (see 4)
7  'not' / 1
8  clause / 5
9  add_calls / 5
   clause / 5
   ! / 0
10 body_calls / 3
   ! / 0
11 functor / 3
   add / 2 (see 2)
12 add_insides / 3
   meta_call_1 / 2
   ! / 0
14 add_inside / 3
   ' / 2
16 call / 1
17 var / 1
18 integer / 1
   ! / 0
   functor / 3
   add / 2 (see 2)
   add_insides / 3 (see 12)

```

---

```

19      meta_call 2 / 3
      body_calls / 3 (see 10)
20      'is' / 2
      add_calls / 5 (see 9)
21      print_calls / 4
      ! / 0
22      start_undefined / 2
23      number_line / 1
24      '<' / 2
      'is' / 2 (see 20)
25      less / 2
26      display / 1
27      tab / 2
      ! / 0
28      wch / 1
      'is' / 2 (see 20)
      tab / 2 (see 27)
29      writeq / 1
      display / 1
30      nl / 0
      'is' / 2 (see 20)
      print_calls / 4 (see 21)
31      start_line / 2
      number_line / 1 (see 23)
      ! / 0
      tab / 2 (see 27)
32      start_unnumbered_line / 1
      display / 1
      tab / 2 (see 27)
33      repetition / 3
      predefined / 2
      ! / 0
      display / 1

```



---

## REFERENCES

---

- Aho, A. V. and Ullman, J. D. (1977). "Principles of Compiler Design." Addison-Wesley, Reading, Massachusetts.
- Astrahan, A. M. (1976). System R: Relational Approach to Database Management. *ACM Trans. Data Base Systems* 1(2), pp. 97-137.
- Ballieu, G. (1983). A Virtual Machine to Implement Prolog. In Pereira *et al.* 1983, pp. 40-52.
- Battani, G. and Méloni, H. (1973). Interpréteur du langage de programmation PROLOG. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.
- Battani, G. and Méloni, H. (1975). Mise en oeuvre des contraintes phonologiques, syntaxiques et sémantiques dans un système de compréhension automatique de la parole. Ph.D. thesis, Université d'Aix-Marseille.
- Bendl, J., Köves, P. and Szeredi, P. (1980). The MPROLOG System. In Tärnlund 1980, pp. 201-209.
- Bergman, M. and Kanoui, H. (1973). Application of Mechanical Theorem-Proving to Symbolic Calculus. In "Proceedings of the 3rd Colloquium on Advanced Computing Methods in Theoretical Physics," Marseille.
- Bergman, M. and Kanoui, H. (1975). Sycophante: Système de calcul formel et d'intégration symbolique sur ordinateur. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.
- Bobrow, D. G. and Wegbreit, B. (1973). A Model and Stack Implementation of Multiple Environments. *Commun. ACM* 16(10), 591-603.
- Bowen, D. L. (1981). DECSys-10 Prolog User's Manual. Department of Artificial Intelligence, University of Edinburgh.
- Bowen, D., Byrd, L. and Clocksin, W. (1983). A Portable Prolog Compiler. In Pereira *et al.* 1983, pp. 74-83.
- Boyer, R. S. and Moore, J. S. (1972). The Sharing of Structure in Theorem Proving Programs. In "Machine Intelligence 7" (B. Meltzer and D. Michie, eds.), pp. 101-116. Edinburgh University Press.
- Bruynooghe, M. (1976). An Interpreter for Predicate Programs: Part 1. Report CW16, Katholieke Universiteit Leuven.
- Bruynooghe, M. (1978). Intelligent Backtracking for an Interpreter of Horn Clause Logic Programs. Report CW16, Katholieke Universiteit Leuven. Also in "Mathematical Logic in Computer Science" (B. Dömölki and T. Gergely, eds.), pp. 215-258. North-Holland Publ., Amsterdam.
- Bruynooghe, M. (1982a). Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs. In Van Caneghem 1982a, pp. 52-55.



- Bruynooghe, M. (1982b). The Memory Management of PROLOG Implementations. In Clark and Tärnlund 1982, pp. 83–98.
- Bruynooghe, M. and Pereira, L. M. (1981). Revision of Top-Down Logical Reasoning through Intelligent Backtracking. Report CIUNL-8/81, Universidade Nova de Lisboa.
- Bundy, A., ed. (1983). "Proceedings of the 8th International Joint Conference on Artificial Intelligence", 8–12 August 1983, Karlsruhe. William Kaufmann, Inc., Los Altos, California.
- Burstall, R. M. and Darlington, J. (1977). Transformation for Developing Recursive Programs. *J. ACM* 24(1), 44–67.
- Campbell, J. A., ed. (1984). "Implementations of PROLOG". Ellis Horwood Ltd., Chichester.
- Chamberlin, D. D., Astrahan, M. M., Eswaran, K. P., Griffith, P. P., Lorie, R. A., Mehl, J. W., Reisner, P. and Wade, B. W. (1976). SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control. *IBM. Res. Dev.* 20(6), pp. 560–575.
- Chomicki, J. and Grudziński, W. (1983). A Database Support System for Prolog. In Pereira *et al.* 1983, pp. 290–303.
- Clark, K. L. (1978). Negation as Failure. In Gallaire and Minker 1978, pp. 293–322.
- Clark, K. L. and Gregory, S. (1983). PARLOG: A Parallel Logic Programming Language. Research Report DOC 83/5, Imperial College, London.
- Clark, K. L. and McCabe, F. G. (1980a). IC-PROLOG—Aspects of Its Implementation. In Tärnlund 1980, pp. 190–197.
- Clark, K. L. and McCabe, F. G. (1980b). IC-PROLOG—Language Features. In Tärnlund 1980, pp. 45–52.
- Clark, K. L. and McCabe, F. G. (1984). "micro-PROLOG. Programming in Logic". Prentice-Hall, Englewood Cliffs, New Jersey.
- Clark, K. L. and Tärnlund, S.-Å. (1977). A First Order Theory of Data and Programs. In "Information Processing 77" (B. Gilchrist, ed.), pp. 939–944. North-Holland, Amsterdam.
- Clark, K. L. and Tärnlund, S.-Å., eds. (1982). "Logic Programming". Academic Press, New York and London.
- Clark, K. L., McCabe, F. G. and Gregory, S. (1979). The Control Facilities of IC-PROLOG. In "Expert Systems in Micro-Electronic Age" (D. Michie, ed.), pp. 129–149. Edinburgh University Press.
- Clark, K. L., Ennals, J. R. and McCabe, F. G. (1982a). "A micro-PROLOG Primer". Logic Programming Associates Ltd., London.
- Clark, K. L., McCabe, F. G. and Gregory, S. (1982b). IC-PROLOG—Language Features. In Clark and Tärnlund 1982, pp. 253–266.
- Clocksin, W. F. and Mellish, C. S. (1981). "Programming in Prolog". Springer-Verlag, Berlin and Heidelberg.
- Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13(6), 377–387.
- Codd, E. F. (1979). Extending the Relational Data Base Model to Capture More Meaning. *ACM Trans. Data Base Systems* 4(4), pp. 397–434.
- Coelho, H. (1982). Man-Machine Communication in Portuguese—A Friendly Library Service System. *Information Systems* 7(2), 163–181.
- Coelho, H., Cotta, J. C. and Pereira, L. M. (1980). How to Solve It in Prolog. Laboratório Nacional de Engenharia Civil, Lisboa.
- Colmerauer, A. (1975). Le grammaires de métamorphose. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.



- Colmerauer, A. (1978). Metamorphosis Grammars. In "Natural Language Communication with Computer" (L. Bolc, ed.), pp. 133-189. Springer-Verlag, Berlin and Heidelberg.
- Colmerauer, A. (1979). Prolog and Infinite Trees. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille. Also in Clark and Tärnlund 1982, pp. 45-66.
- Colmerauer, A. (1982). PROLOG II. Manuel de référence et modèle théorique. Groupe d'Intelligence Artificielle, Université Marseille II.
- Colmerauer, A. (1983). Prolog in Ten Figures. In Bundy 1983, pp. 487-499.
- Colmerauer, A., Kanoui, H., Roussel, P. and Pasero, R. (1972). Un système de communication homme-machine en français. Rapport préliminaire. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.
- Colmerauer, A., Kanoui, H., Roussel, P. and Pasero, R. (1973). Un système de communication homme-machine en français. Rapport de recherche sur le contrat CRI no 72-18 de février 72 à juin 73. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.
- Colmerauer, A., Kanoui, H. and Van Caneghem, M. (1979). Etude et réalisation d'un système Prolog. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.
- Colmerauer, A., Kanoui, H. and Van Caneghem, M. (1981). Last Steps toward an Ultimate Prolog. In "Proceedings of the 7th International Joint Conference on Artificial Intelligence" (R. Schank, ed.), Vancouver, Canada, pp. 947-948.
- Colmerauer, A., Kanoui, H. and Van Caneghem, M. (1983). Prolog, bases théoriques et développements actuels. *Techniques et Science Informatiques* 2(4), 271-311. English translation in *Technology and Science of Informatics* 2(4).
- Conery, J. S. and Kibler, D. F. (1983). AND Parallelism in Logic Programs. In Bundy 1983, pp. 539-543.
- Dahl, V. (1977). Un système déductif d'interrogation de banques de données en espagnol. Ph.D. thesis, Université d'Aix-Marseille.
- Dahl, V. (1980). Two Solutions for the Negation Problem. In Tärnlund 1980, pp. 61-72.
- Date, C. J. (1982). "An Introduction to Database Systems", 3rd. ed., Vol. 1. Addison-Wesley, Reading, Massachusetts.
- Dijkstra, E. W. (1975). Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 453-457.
- Donz, P. (1979). Une méthode de transformation et d'optimisation de programmes Prolog: définition et implémentation. Ph.D. thesis, Université d'Aix-Marseille.
- Eisinger, N., Kasif, B. and Minker, J. (1982). Logic Programming—a Parallel Approach. In Van Caneghem 1982a, pp. 71-77.
- Emden, M. H. van (1981). AVL-Tree Insertion: A Benchmark Program Biased towards Prolog. *Logic Programming Newslett.* 2, 4.
- Emden, M. H. van (1982). An Interpreting Algorithm for Prolog Programs. In Van Caneghem 1982a, pp. 56-64. Also in Campbell 1984, pp. 93-110.
- Emden, M. H. van and Kowalski, R. A. (1979). The Semantics of Predicate Logic as a Programming Language. *J. ACM* 23(4), 733-744.
- Ennals, J. R. (1983). "Beginning Micro-Prolog". Ellis Horwood, Chichester, U.K.
- Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem-Proving to Problem Solving. *Artif. Intell.* 2, 235-246.
- Filgueiras, M. (1982). A Prolog Interpreter Working with Infinite Terms. Report FCT/UNL-20/82, Universidade Nova de Lisboa. Also in Campbell 1984, pp. 250-258.
- Filgueiras, M. and Pereira, L. M. (1983). Relational Databases a La Carte. In Pereira *et al.* 1983, pp. 389-407.
- Gallaire, H. (1983). Logic Databases versus Deductive Databases. In Pereira *et al.* 1983, pp. 608-622.



- Gallaire, H. and Minker, J., eds. (1978). "Logic and Databases". Plenum Press, New York.
- Gallaire, H., Minker, J. and Nicolas, J.-H., eds. (1981). "Advances in Database Theory", vol. I. Plenum Press, New York.
- Gregory, S. (1980). Towards the Compilation of Annotated Logic Programs. Research Report DOC 80/16, Imperial College, London.
- Gries, D. (1971). "Compiler Construction for Digital Computers". Wiley and Sons, New York.
- Guizol, J. (1975). Synthèse du français à partir d'une représentation en logique du premier ordre. Ph.D. thesis, Université d'Aix-Marseille.
- Guizol, J. and Méloni, H. (1976). Prolog modulaire. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.
- Hill, R. (1974). LUSH Resolution and Its Completeness. DCL Memo 78, University of Edinburgh.
- Hoare, C. A. D. (1962). Quicksort. *Comput. J.* 5(1), 10–15.
- Hogger, C. J. (1979). Derivation of Logic Programs. Ph.D. thesis, Imperial College, London.
- Joubert, M. (1974). Un système de résolution de problèmes a tendance naturelle. Ph.D. thesis, Université d'Aix-Marseille.
- Kanoui, H. (1973). Application de la démonstration automatique aux manipulations algébrique et à l'intégration formelle sur ordinateur. Ph.D. thesis, Université d'Aix-Marseille.
- Kanoui, H. (1982). PROLOG II. Manuel d'exemples. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II.
- Kanoui, H. and Van Caneghem, M. (1980). Implementing a Very High Level Language on a Very Low Cost Computer. In "Information Processing 80" (S. Lavington, ed.), pp. 349–354. North-Holland, Amsterdam.
- Kluźniak, F. (1981). Remarks on Coroutines in Prolog. In Szpakowicz 1981, pp. 19–29.
- Kluźniak, F. (1984). The "Marseille Interpreter"—a Personal Perspective. In Campbell 1984, pp. 65–70.
- Kluźniak, F. and Szpakowicz, S. (1983). "Prolog" WNT (Wydawnictwa Naukowo-Techniczne), Warsaw.
- Kluźniak, F. and Szpakowicz, S. (1984). Prolog—a Panacea? In Campbell 1984, pp. 71–84.
- Knuth, D. E. (1968). Semantics of Context-Free Languages. *Math. Syst. Theory* 2, 127–145.
- Komorowski, H. J. (1982). QLOG—The Programming Environment for Prolog in LISP. In Clark and Tärnlund 1982, pp. 315–322.
- Koster, C. H. A. (1974). Using the CDL Compiler—Compiler. In "Compiler Construction. An Advanced Course" (F. J. Bauer and J. Eickel, eds.), pp. 366–426. Lecture Notes in Computer Science 21, Springer-Verlag, Berlin and Heidelberg.
- Kowalski, R. A. (1972). The Predicate Calculus as a Programming Language. In "Proceedings of the International Symposium and Summer School on Mathematical Foundations of Computer Science", Jabłonna near Warsaw, Poland.
- Kowalski, R. A. (1974). Predicate Logic as Programming Language. In "Proceedings of the IFIP Congress", pp. 569–574. North-Holland, Amsterdam.
- Kowalski, R. A. (1978). Logic for Data Description. In Gallaire and Minker 1978, pp. 77–103.
- Kowalski, R. A. (1979a). Algorithm = Logic + Control. *Commun. ACM* 22, 424–431.
- Kowalski, R. A. (1979b). "Logic for Problem Solving". North-Holland, Amsterdam.
- Kowalski, R. A. and Kuehner, D. (1971). Linear Resolution with Selection Function. *Artificial Intelligence* 2(3/4), 227–260. Also in "The Automation of Reasoning II" (J. H. Siekmann and G. Wrightson, eds.) Springer-Verlag, Berlin and Heidelberg, 1983.



- Lloyd, J. W. (1982). An Introduction to Deductive Database Systems. Department of Computer Science Report TR 81/3, University of Melbourne.
- McCabe, F. G. (1981). "Micro PROLOG Programmer's Reference Manual". Logic Programming Associates Ltd., London.
- Mellish, C. S. (1981). Automatic Generation of Mode Declarations in Prolog Programs. Paper presented at Workshop on Logic Programming, Long Beach, Los Angeles, California.
- Mellish, C. S. (1982). An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter. In Clark and Tärnlund 1982, pp. 99–106.
- Mellish, C. and Hardy, S. (1983). Integrating Prolog into the Poplog Environment. In Bundy 1983, pp. 533–535. Also in Campbell 1984, pp. 147–162.
- Moss, C. D. S. (1979). A New Grammar for Algol 68. Department of Computing report 79/6, Imperial College, London.
- Mycroft, A. and O'Keefe, R. (1983). A Polymorphic Type System for Prolog. In Pereira *et al.* 1983, pp. 107–122.
- Neves, J. and Williams, M. (1983). Towards a Co-operative Data Base Management System. In Pereira *et al.* 1983, pp. 341–370.
- Neves, J., Anderson, S. and Williams, M. (1983). Security and Integrity in Logic Data Bases using QBE. In Pereira *et al.* 1983, pp. 304–340.
- Pasero, R. (1973). Représentation du français en logique du 1er ordre en vue de dialoguer avec un ordinateur. Ph.D. thesis, Université d'Aix-Marseille.
- Pereira, L. M. and Porto, A. (1980a). An Interpreter for Logic Programs using Selective Backtracking. Report 3/80, Centro de Informatica da Universidade Nova de Lisboa.
- Pereira, L. M. and Porto, A. (1980b). Selective Backtracking for Logic Programs. In "Proceedings of the 5th Conference on Automated Deduction" (W. Bibel and R. Kowalski, eds.), pp. 306–317. Springer-Verlag, Berlin and Heidelberg.
- Pereira, L. M. and Porto, A. (1981). All Solutions. *Logic Programming Newslett.* 2, 9–10.
- Pereira, L. M. and Porto, A. (1982). Selective Backtracking. In Clark and Tärnlund 1982, pp. 107–114.
- Pereira, F. C. N. and Warren, D. H. D. (1980). Definite Clause Grammars for Language Analysis—a Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artif. Intell.* 13(3), 231–278.
- Pereira, L. M., Pereira, F. C. N. and Warren, D. H. D. (1978). User's Guide to DECSys-tem-10 Prolog (Provisional Version). Department of Artificial Intelligence, University of Edinburgh.
- Pereira, L. M., Porto, A., Monteiro, L. and Filgueiras, M., eds. (1983). "Logic Programming Workshop '83 Proceedings", Praia da Falésia, Algarve, Portugal, 26 June to 1 July, 1983, Universidade Nova de Lisboa.
- Porto, A. (1982). EPILOG: A Language for Extended Programming in Logic. In Van Caneghem 1982a, pp. 31–37.
- Robinson, J. A. (1965). A Machine-oriented Logic Based on the Resolution Principle. *J. ACM* 12(1), pp. 23–41.
- Robinson, J. A. (1979). "Logic: Form and Function—the Mechanization of Deductive Reasoning". North-Holland, Amsterdam.
- Roussel, P. (1975). PROLOG, manuel de référence et d'utilisation. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.
- Sedgewick, R. (1983). "Algorithms". Addison-Wesley, Reading, Massachusetts.
- Sergot, M. (1982). A Query-the-User Facility for Logic Programming. Research report DOC 82/18, Imperial College, London.



- Shapiro, E. Y. (1983a). "Algorithmic Program Debugging". MIT Press, Cambridge, Massachusetts.
- Shapiro, E. Y. (1983b). A Subset of Concurrent Prolog and Its Interpreter. ICOT (Institute for New Generation Computer Technology) Technical Report TR-003, Tokyo.
- Stonebraker, M., Wong, E., Kreps, P. and Held, G. (1976). The Design and Implementation of INGRES, *ACM Trans. Data Base Systems* 1(3), pp. 189–222.
- Szeredi, P. (1977). PROLOG—a Very High Level Language Based on Predicate Logic. Preprints of 2nd Hungarian Computer Science Conference, Budapest.
- Szeredi, P. (1982). Module Concept for Prolog. Paper presented at the Prolog Programming Environments Workshop, Linköping.
- SzKI (1982). MPROLOG User's Manual. Szamistastechnikai Koordinacios Intezet, Budapest.
- Szpakowicz, S., ed. (1981). Papers in Logic Programming I. IInfUW Report No. 104, Warsaw University.
- Tärnlund, S.-Å., ed. (1980). Preprints of: Logic Programming Workshop, 14–16 July 1980, Debrecen, Hungary.
- Ullman, J. D. (1982). "Principles of Database Systems", 2nd ed. Computer Science Press, Rockville, Maryland.
- Van Caneghem, M., ed. (1982a). "Proceedings of the 1st International Logic Programming Conference", 14–17 September 1982, Faculté de Sciences de Luminy, Marseille, France.
- Van Caneghem, M. (1982b). PROLOG II. Manuel d'utilisation. Groupe d'Intelligence Artificielle, Université Marseille II.
- Vasey, P. (1982). AVL-Tree Insertion Revisited. *Logic Programing Newslett.* 3, 11.
- Warren, D. H. D. (1974). WARPLAN—a System for Generating Plans. DGL Memo 76, University of Edinburgh.
- Warren, D. H. D. (1976). Generating Conditional Plans and Programs. In "Proceedings of the AISB Summer Conference", Edinburgh, pp. 344–354.
- Warren, D. H. D. (1977a). Implementing Prolog—Compiling Predicate Logic Programs. DAI Report Nos. 39 and 40, University of Edinburgh.
- Warren, D. H. D. (1977b). Logic Programming and Compiler Writing. DAI Report No. 44, University of Edinburgh.
- Warren, D. H. D. (1980a). An Improved Prolog Implementation Which Optimises Tail Recursion. In Tärnlund 1980, pp. 1–11.
- Warren, D. H. D. (1980b). Logic Programming and Compiler Writing. *Software—Practice and Experience* 10(2), 97–125.
- Warren, D. H. D. (1981). Efficient Processing of Interactive Relational Database Queries Expressed in Logic. In "Proceedings of the 7th International Conference on Very Large Data Bases", Cannes, pp. 272–281.
- Warren, D. H. D. and Pereira, F. C. N. (1982). An Efficient Easily Adaptable System for Interpreting Natural Language Queries. *Am. J. Computational Linguistics* 8(3–4), 110–119.
- Warren, D. H. D., Pereira, L. M. and Pereira, F. C. N. (1977). Prolog—the Language and Its Implementation Compared with Lisp. Presented at the ACM Symposium on Artificial Intelligence and Programming Languages, Rochester, New York. SIGART Newsletter No. 64, *SIGPLAN Notices* 12(8), 1977, pp. 109–115.
- Wijngaarden, van A., ed. (1976). "Revised Report on the Algorithmic Language Algol 68". Springer-Verlag, Berlin and Heidelberg.



- Wirth, N. (1976), "Algorithms + Data Structures = Programs". Prentice-Hall, Englewood Cliffs, New Jersey.
- Wise, M. J. (1984). EPILOG: Re-interpreting and Extending Prolog for a Multiprocessor Environment. In Campbell 1984, pp. 341-351.
- Zloof, M. M. (1977). Query-by-Example: A Data Base Language. *IBM Syst. J.* **16**(4), 324-342.

The subject index was originally compiled from the first edition of this volume. Therefore the page numbers for the following entries should now read

Chat80, 237–238  
Coroutining, 57  
    in Prolog, 251–252

Databases, relational, 226–237  
Dialects  
    micro-Prolog and MPROLOG, 253–254  
    Prolog I, 249–250  
    Prolog II, 250–253

Infinite trees, 23, 250

micro-Prolog, 253  
Modularisation, in Prolog, 252, 254  
MPROLOG, 253–254

Planning, 215–226  
Prolog I, 249–250  
Prolog II, 250–253

Query-by-Example, 231

Sequel, 231  
Simple, 253

Toy-sequel, 231–247  
Tree(s)  
    infinite, 23, 250

WARPLAN, 216–226



# INDEX

Page numbers in *italics* indicate material referring to implementation issues; page numbers followed by *n* indicate material in footnotes.

## A

Alternatives, in grammar rules, 79–81  
 Anonymous variables, 11  
 Arguments, 4  
 Arity, 4–5  
 Array analogues, in Prolog, 113–116  
 Associative functor, 155  
 Atoms, in Toy-Prolog interpreter, 189–190  
 Attribute grammars, 82, 84  
 Axioms, 44

## B

Backtracking, 26–27  
   how to use, 28–32  
   intelligent, 57  
   selective, 57  
 Binary trees, representation in Prolog, 89–94  
 Bound variables, 9, 18  
 Built-in procedures, *see also* Procedure(s),  
   built-in  
   ! / 0, 36, 121–124, 163  
   , / 2, 39, 150  
   ; / 2, 39, 150  
   \=/ / 2, 153  
   </2, 28, 152  
   =../ 2, 89, 117, 160  
   = / 2, 28–29, 152  
   =\=/ / 2, 152

=:=/ 2, 28, 152  
 =</ 2, 28, 152  
 ==/ 2, 153  
 =\=/ / 2, 28, 152  
 > / 2, 28, 152  
 >= / 2, 28, 152  
 @</ 2, 28, 152  
 @=</ 2, 28, 152  
 @>/ 2, 28, 152  
 @>= / 2, 28, 152  
 abolish / 2, 162  
 alphanum / 1, 158  
 ancestor / 1, 164  
 arg / 3, 160  
 assert / 1, 109, 129, 161  
 assert / 3, 161  
 asserta / 1, 161  
 assertz / 1, 161  
 atom / 1, 159  
 bagof / 3, 112, 166  
 bigletter / 1, 158  
 bracket / 1, 158  
 call / 1, 118, 163  
 check / 1, 150  
 clause / 2, 162  
 clause / 5, 161  
 consult / 1, 16, 162  
 debug / 0, 165  
 delop / 1, 156  
 digit / 1, 158  
 display / 1, 154

echo / 0, 154  
 eqvar / 2, 152  
 error / 1, 147–148  
 fail / 0, 31, 149  
 functor / 3, 159  
 halt / 1, 164  
 integer / 1, 28, 158  
 is / 2, 13, 151  
 isclosedlist / 1, 165  
 iseoln / 1, 157  
 lastch / 1, 157  
 length / 2, 165  
 less / 2, 151  
 letter / 1, 158  
 listing / 0, 163  
 listing / 1, 163  
 member / 2, 166  
 nl / 0, 13, 157  
 nodebug / 0, 165  
 noecho / 0, 154  
 nonexistent / 0, 165  
 nononexistent / 0, 165  
 nonvarint / 1, 159  
 not / 1, 39, 122, 125, 149  
 numbervars / 1, 166  
 once / 1, 123, 150  
 op / 3, 8, 155  
 ordchr / 2, 157  
 phrase / 2, 69, 120, 165  
 pname / 2, 73, 159  
 pnamei / 2, 159  
 predefined / 2, 162  
 prod / 4, 151  
 protect / 0, 162  
 rch / 0, 156  
 rdch / 1, 157  
 rdchsk / 1, 157  
 read / 1, 155  
 reconsult / 1, 16, 162–163  
 redefine / 0, 162  
 repeat / 0, 163  
 retract / 1, 109, 129, 161–162  
 retract / 3, 161  
 see / 1, 153  
 seeing / 1, 153  
 seen / 0, 154  
 side\_effects / 1, 150  
 skipbl / 0, 156  
 smalletter / 1, 158  
 solochar / 1, 158

status / 0, 157  
 stop / 0, 164  
 sum / 3, 150  
 symch / 1, 158  
 tag / 1, 164  
 tagcut / 1, 164  
 tagexit / 1, 164  
 tagfail / 1, 164  
 tell / 1, 154  
 telling / 1, 154  
 told / 0, 154  
 true / 0, 149  
 var / 1, 158  
 wch / 1, 157  
 write / 1, 13, 154–155  
 writeq / 1, 155

## C

Call(s)  
   order of, 34–35  
   variable, 38–40  
 Canonic form, 6  
 Characters, 8  
 Chat80, 252–253  
 Choice points, 27  
 Clausal representation, of data structures,  
   107–113  
 Clause(s), 23–24, 46, 144  
   as global data, 129–130  
   Horn, 46–50  
   order of, 34–35  
   unit, 32, 47, 144  
 Closed lists, 100  
 Command, 15, 16, 144  
 Command mode, 15  
 Comment, 16, 146  
 Compound objects, 3  
   descriptions, 5–8  
   functors, 3–5  
   strings, 8  
 Conditions, in grammar rules, 73–76, 144  
 Connectives, logical, 42  
 Consequence, logical, 144  
 Constants, 1–3  
 Context, in grammar rules, 76–79  
 Control, 54, 163–164, 178–179  
   backtracking, 26–32  
   cut, 35–38  
   general form of a procedure, 24–26



order of calls and clauses, 34–35  
 static interpretation of procedure, 32–33  
 in Toy-Prolog interpreter, 194–199  
 variable calls, 38–40  
 Convenience, built-in procedures, 149–150  
 Copy stack, 175–176  
 Coroutining, 57  
   in Prolog, 257–258  
 Current resolvent, 47  
 Cut procedure, 35–38, *see also* Built-in  
   procedures, 1/0

## D

DAGs, *see* Directed acyclic graphs  
 Data bases, relational, 228–253  
 Data structures, 88  
   access to structure of terms, 116–120  
   array analogues, 113–116  
   clausal representation of, 107–113  
   compound objects, 3–8  
   constants, 1–3  
   difference lists, 104–107  
   open trees and lists, 98–103  
   simple trees and lists, 88–98  
   terms, 9–11  
   variables, 8–9  
 Debugging, 164–165  
 Definite clause grammars, 68n  
 Definition mode, 15  
 Dialects  
   micro-Prolog and MPROLOG, 259–260  
   Prolog I, 255–256  
   Prolog II, 256–259  
 Dictionary, 189–190  
 Difference lists, 104–107  
 Directed acyclic graphs (DAGs), 11, 168  
 Directives, 15–17, 144

## E

Efficiency, in grammar rules, 81–83  
 Euler paths, 139–142  
 Extralogical features, of Prolog, 35–38

## F

Factorization, 82  
 Fail point(s), 27, 178–179

Fail point record, 178–179  
 Failure, 27  
   forced, 125  
   as programming tool, 125–129  
 Formal reasoning, 44–45  
 Formulae, 41–42  
   inconsistent, 43  
   interpretation of, 42–44  
   tautology, 43  
   true (false) in a model, 43  
   true (false) in an interpretation,  
     43  
 Free variable, 9  
 Frozen variable, 179  
 Fullstop, 15, 146  
 Functor, 3–5, 42, 145  
   associative, 155  
   infix, 6, 155  
   left-associative, 6, 155  
   main, 5  
   non-associative, 155  
   postfix, 6, 155  
   prefix, 6, 155  
   priority, 7  
   right-associative, 6, 155

## G

Generator, 31  
 Global stack, 176  
 Global variables, 176  
 Goal statements, 16  
 Grammar(s)  
   attribute, 82, 84  
   definite clause, 68n  
   metamorphosis, 68  
   two-level, 82  
 Grammar preprocessor, 212–213  
 Grammar processing, 165  
 Grammar rules, 144  
   extensions  
     alternatives, 79  
     conditions, 73–76  
     context, 76–79  
   in Prolog, 68  
   simplest form of, 67–69  
 Ground variable, 9

## H

Horn clauses, 46–50

**I**

- Identifier, 2
- Inconsistency, 43
- Indexing, 109–110
- Inference rules, 44
- Infinite trees, 23, 256
- Infix functor, 6, 155
- Initialisation, 201
- Input / output
  - listing control, 154
  - single characters, 156–157
  - switching streams, 153–154
  - terms, 154–156
- Instantiation
  - of a term, 10–11
  - of a variable, 9
- Integer(s), 2, 145
  - comparing, 151–152
- Interpretation
  - of formulae, 42–44
  - static, of procedures, 32–33

**L**

- Language, intermediate, 201–203
- Left-associative functor, 6, 155
- Library, 213–214
- Linear lists, representation in Prolog, 95
- List(s), 145
  - closed, 100
  - difference, 104–107
  - linear, 95
  - open, 98–103
  - representation in Prolog, 7–8
  - reversal of, 131–134
  - simple, 88–98
- List notation, 95
- Literals, 46
  - negative, 46
  - positive, 46
- Local stack, 176
- Local variables, 176
- Logical connectives, 42
- Logical consequence, 44

**M**

- Main functor, 5
- Matching, *see* Unification

- Metamorphosis grammars, 68
- micro-Prolog, 259
- Mixed operators, 7, 155, 156
- Mode declarations, 177
- Model, 43
  - true (false) formulae in, 43
- Modularisation, in Prolog, 258, 260
- Molecules, 169*n*
- Monitor, 203
- MPROLOG, 259–260

**N**

- Name(s)
  - comparing, 151–152
  - quoted, 2
  - variable, 9
- Negation, in Prolog, 38
- Negative literals, 46
- Nondeterminism, in Prolog, 31
- Non-associative functor, 155
- Non-structure sharing (NSS), 168, 172–176
- Nonterminal (symbol), 68, 144
- Nonterminal symbols, parameters of, 69–73
- NSS, *see* Non-structure sharing

**O**

- Occur check, 23
- Open lists, 98–103
- Operations, 11–13, *see also* Clause(s)
  - directives, 15–17, 144
  - simplest form of procedure, 13–15
  - unification, 17–23
- Operator(s), 7
  - mixed, 7
  - predefined, 148–149
  - type of, 155

**P**

- Parallelism 57
- Parsing problem, representation of, 59–67
- Planning, 215–228
- Positive literals, 46
- Postfix functor, 6, 155
- Predefined operators, 148–149
- Predicate(s), 42
- Predicate symbol, 12, 42



Prefix functor, 6, 155  
 Priority, 7, 155  
 Procedure(s), 11–13, *see also* Built-in  
   procedures  
     accessing, 160–163  
     built-in, 12, 147–149  
     predefined, 147  
     side-effects of, 27  
     system, 147  
   general form of, 24–26  
   nondeterministic, 31  
   simplest form of, 13–15  
   static interpretation of, 32–33  
   system, 12, 147, 199  
   in Toy-Prolog interpreter, 189, 190, 199  
 Procedure body, 23, 144  
 Procedure call, 144  
 Procedure definition, 144  
 Procedure heading, 14, 144  
 Prolog I, 255–256  
 Prolog II, 256–259  
 Prolog-10, 143  
 Prolog-10 character input  
   get, 156  
   get0, 156  
   put, 156  
   skip, 156  
 Prolog grammar rule, 68  
 Proof, 44  
 Proof tree, 52  
 Prototype(s), 169, 190–194

## Q

Quantifiers, 42  
   scope of, 42  
 Query, 15–16, 144  
 Query-by-Example, 233  
 Quoted name, 2, 145

## R

Reader, for interpretation of Prolog-10 in  
   Toy-Prolog, 203–212  
 Reasoning, formal, 44–45  
 Recursion, elimination of, in metamorpho-  
   sis grammars, 83–85  
 Reductio ad absurdum, 45  
 Resolution, rule of, 45–46  
 Resolvent, current, 25–26

Right-associative functor, 6, 155  
 Rule of resolution, 45–46

## S

Scope, of quantifiers, 42  
 Search space, 51  
 Sequel, 233  
 Side-effect, of built-in procedure, 27  
 Simple, 259  
 Simple lists, 88–98  
 Skeletons, 169*n*  
 Sorting, 134–138  
 Stack  
   copy, 175–176  
   global, 176  
   local, 176  
 Storage areas, 187–188  
 Strategy, 51–58  
 Strings, 8, 145  
 Structure sharing, 168–172, 176–178  
 Switching streams, 153–154  
 Symbol, 2, 145  
 Syntax of Prolog, 143–147  
 System procedures, 12  
   built-in, 147  
   predefined, 147  
   in Toy-Prolog interpreter, 199

## T

Tail recursion optimisation (TRO), 179–182  
   delayed, 182  
 Tautology(ies), 43  
 Term(s), 5, 9–11, 42, 144  
   accessing structure of, 159–160  
   instantiation of, 10–11  
 Term descriptions, 168  
 Term handle, 169  
 Terminal (symbol), 68, 74, 144  
 Term instances, 168, 190–194  
 Theorems, 44  
 Theory, 44  
 Toy-Sequel, 233–253  
 Trail, 179  
 Translator, 214  
 Tree(s)  
   binary, 89–94  
   closed, 101  
   infinite, 23, 256

open, 98–103  
 proof, 52  
 simple, 88–98  
 TRO, *see* Tail recursion optimisation  
 Two-level grammars, 82

## U

Unification, 17–23  
 success and failure, 17  
 Unit clause, 32, 47, 144  
 Univ, *see* Built-in procedures, =../2  
 Universe, 43  
 user (filename of the user's terminal), 16,  
 153

## V

Variables(s), 8–9, 42, 145  
 anonymous, 11

bound, 9  
 bound together, 18  
 free, 9  
 frozen, 179  
 global, 176  
 ground, 9  
 instantiation of, 9  
 local, 176

Variable binding, 9  
 Variable calls, 38–40  
 Variable frame, 169  
 Variable name, 9  
 Variable terminal (symbol), 74

## W

WARPLAN, 216–228



A.P.I.C. Studies in Data Processing  
*General Editors: Fraser Duncan and M. J. R. Shave*

1. Some Commercial Autocodes. A Comparative Study\*  
E. L. Willey, A. d'Agapeyeff, Marion Tribe, B. J. Gibbens  
and Michelle Clarke
2. A Primer of ALGOL 60 Programming  
E. W. Dijkstra
3. Input Language for Automatic Programming\*  
A. P. Yershov, G. I. Kozhukhin and U. Voloshin
4. Introduction to System Programming\*  
Edited by Peter Wegner
5. ALGOL 60 Implementation. The Translation and Use of  
ALGOL 60 Programs on a Computer  
B. Randell and L. J. Russell
6. Dictionary for Computer Languages\*  
Hans Breuer
7. The Alpha Automatic Programming System\*  
Edited by A. P. Yershov
8. Structured Programming  
O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare
9. Operating Systems Techniques  
Edited by C. A. R. Hoare and R. H. Perrott
10. ALGOL 60. Compilation and Assessment  
B. A. Wichmann
11. Definition of Programming Languages by Interpreting Automata  
Alexander Ollongren
12. Principles of Program Design  
M. A. Jackson
13. Studies in Operating Systems  
R. M. McKeag and R. Wilson
14. Software Engineering  
R. J. Perrott

(continued)

\*Out of print.

15. Computer Architecture: A Structured Approach  
R. W. Doran
16. Logic Programming  
Edited by K. L. Clark and S.-A. Tärnlund
17. Fortran Optimization  
Michael Metcalf
18. Multi-microprocessor Systems  
Y. Paker
19. Introduction to the Graphical Kernel System—GKS  
F. R. A. Hopgood, D. A. Duce, J. R. Gallop and  
D. C. Sutcliffe
20. Distributed Computing  
Edited by Fred B. Chambers, David A. Duce and  
Gillian P. Jones
21. Introduction to Logic Programming  
Christopher John Hogger
22. Lucid, the Dataflow Programming Language  
William W. Wadge and Edward A. Ashcroft
23. Foundations of Programming  
Jacques Arsac
24. Prolog for Programmers  
Feliks Kluźniak and Stanisław Szpakowicz



## PROLOG FOR PROGRAMMERS

### From the reviews

*"It contains material I have not been able to find in any other book — material that is a significant part of the Prolog 'culture'. It offers an excellent discussion of metamorphosis grammars and the best treatment of data structures that I have encountered. . . . It is in many ways a wonderful book."*

Overbeek's Outlook, June 1986

*"A high competence practical guide for the Prolog programmer. . . . it should be on everyone's desk."*

Zentralblatt fur Mathematik, 1986

This is a self-contained handbook of advanced logic programming. It assumes the reader is not a novice to computer science, has perhaps read an introductory book on Prolog and would now like to do some non-trivial work using the language.

A thorough discussion of Prolog treatment of data structures, an overview of Prolog's logical foundations, a concise but comprehensive presentation of logic grammars, simple and advanced programming techniques, style and efficiency guidelines, a real exercise in Prolog implementation, and two non-trivial case studies make this an invaluable handbook for both professional and student. Experienced Prolog programmers may find the systematic exposition helpful, and the discussion of implementation issues instructive.

One of the unique features of the book is its detailed coverage of Metamorphosis (or Definite Clause) Grammars: programmers applying Prolog to natural or formal language processing will definitely appreciate this.

A small but complete interpreter is provided on the disk in source form (Pascal and Prolog) so that readers can use it to run their programs or take it apart to study in detail to achieve an in-depth understanding of Prolog.

### Academic Press

Harcourt Brace Jovanovich, Publishers

LONDON ORLANDO SAN DIEGO  
NEW YORK AUSTIN BOSTON SYDNEY  
TOKYO TORONTO

---

Academic Press Inc. (London) Ltd.

24/28 Oval Road, London NW1 7DX, England

---

Academic Press, Inc., Orlando, Florida 32887

---

Academic Press Canada

55 Barber Greene Road, Don Mills, Ontario M3C 2A1, Canada

---

Academic Press Australia

P.O. Box 300, North Ryde, N.S.W. 2113, Australia

---

Academic Press Japan, Inc.

Iidabashi Hokoku Bldg., 3-11-13, Iidabashi, Chiyoda-ku, Tokyo 102, Japan